Weak References

Author: John McCall

Date: 2013-05-23

Abstract: This paper discusses the general concept of weak references, including various designs in other languages, and proposes several new core language features and a more sophisticated runtime support feature which can be exploited in the standard library.

Warning

This document was used in planning Swift 1.0; it has not been kept up to date and does not describe the current or planned behavior of Swift.

Reference Graphs

Let's run through some basic terminology.

Program memory may be seen abstractly as a (not necessarily connected) directed graph where the nodes are objects (treating transient allocations like local scopes as objects) and the edges are references. (Multi-edges and self-edges are permitted, of course.)

We may assign a level of strength to each of these edges. We will call the highest level *strong*; all others are some flavor of *weak*.

Every object has a *strength of reference* which arises from the reference graph. This can be any level of strength or the special value *unreferenced*. The strength of a path is the minimum strength of any edge in the path. The strength of a set of paths is the maximum strength of all the paths, unless the set is empty, in which case it is *unreferenced*.

In general, the implementation is only outright forbidden to deallocate an object if it is strongly referenced. However, being somehow weakly referenced may trigger some sort of additional guarantee; see the Language Precedents section.

In a cycle-collecting environment, certain nodes are given special treatment as *roots*; these nodes are always strongly referenced. Otherwise, the strength of reference for an object is the strength of all paths from any root to the object.

In a non-cycle-collecting environment, the strength of reference for an object is the strength of all the direct references to that object, taken as length=1 paths. Note that this environmental consideration becomes a language guarantee: even if the implementation can trivially prove that an object is referenced only by itself, it is still not permitted to deallocate the object.

It is common for certain kinds of reference to not receive a full guarantee. For example, a strong reference from a local variable may lose effectiveness as soon

as the variable is no longer needed (but before it formally leaves scope). This is pervasive in GC environments but also true in e.g. ObjC ARC.

In some programs, especially event-driven programs like UIs or servers, it can be useful to consider the *static* reference graph as captured during a notional steady state. Local scopes may form many references to objects in the static graph, but unless the scope persists indefinitely or a major restructuring is triggered, these references are likely to have no effect on the reference graph, and so their strength of reference has no semantic importance. Understanding this helps to explain why many environments provide no direct way to form a local weak reference.

Language and Library Precedents

We're only going to discuss *managed* precedents here.

It is possible to create a kind of weak reference by just not providing any special behavior to an object reference; if the object is deallocated, the reference will dangle and uses of it will likely crash or cause corruption. This could happen by e.g. declining to insert reference-count manipulations for a particular variable (in an ARC-like environment) or not mapping a variable's frame offset as a pointer in a type map (in a precise-GC environment). We will call this *dangling weak*.

Objective-C

All modes of Objective-C automate memory management for synthesized properties. In GC and ARC, accessors just use the normal semantics for the underlying ivar (plus copy/atomic guarantees, of course). In MRC, accessors use dangling weak semantics unless they're `retain` or `copy`, in which case they maintain a +1 refcount invariant on the referent.

In GC and ARC, variables qualified with `__weak` are immediately zeroed out when the referenced object begins deallocation. There is no syntactic difference on use; it's just possible that the value read will be `nil` instead of whatever was last written there, possibly causing the loading code to crash (e.g. on an ivar access) or silently do nothing (e.g. on a message send). There is an opt-in warning in ARC for certain uses of values loaded from `__weak` ivars.

In GC, it is also possible to construct a dangling weak reference by storing an object pointer into (1) unscanned heap memory or (2) an instance variable that's not of object-pointer type and isn't qualified with `__strong` or `__weak`. Otherwise, object references are strong (including all references from local scopes).

In ARC, it is possible to construct a dangling weak reference by using the `__unsafe_unretained` qualifier or by bridging a pointer value to a C pointer type.

C++

C++ smart pointers (e.g. `std::unique_ptr`) typically permit the creation of a dangling-weak reference by providing an accessor to get the pointer as a normal C pointer. (Even if they didn't have `get()`, you could manually call `operator->` to get the same effect.)

C++'s `std::shared_ptr` permits the formation of weak pointers (`std::weak_ptr`) from shared pointers. It is not possibly to directly use a weak pointer; it must first be converted back to a `shared_ptr`, either by using the `lock()` operation (which produces a null pointer if the referent has been deallocated) or by directly constructing a `shared_ptr` with the `weak_ptr` (which throws an exception if the referent has been deallocated). There is also a way to explicitly query whether a `weak_ptr` is still valid, which may be more efficient than checking the result of the cast.

Java

Java does not provide any facility for dangling weak references. The standard library does provide three levels of weak reference (in `java.lang.ref`). References cannot be re-seated (although they can be explicitly cleared), and users must call `get()` in order to access the value, which may yield `null`.

There is a great deal of interesting discussion of these reference classes here.

Java `Reference` objects may be constructed with an optional `ReferenceQueue`; if so, then when the object's reachability changes, the reference object will be added to that queue. This permits data structures to clean up after cleared soft references without needing to either periodically scan the entire structure or be fully lazy. Additional data may be added to the reference object by subclassing it.

The references are presented in order of decreasing strength.

`SoftReference` is a sort of quasi-strong reference which holds onto the object until the VM begins to run out of memory. Soft references to softly-referenced objects are guaranteed to have been cleared before the VM can throw an `OutOfMemoryError`. The reference will be cleared before it is added to its reference queue (and so the reference queue cannot resurrect the object). The intent of soft references is to enable memory-sensitive caches, but in practice a memory-sensitive cache would probably want to implement a more subtle replacement strategy than "drop things at random as soon as memory runs low". A more interesting use is a memory-guided circuit-breaker: when building up a very large structure, hold it in a soft reference, and if that references goes null during construction, just bail out. But that's a pretty tricky use-case to get right.

`WeakReference` is intended for use in non-memory-sensitive weak caches, like a uniquing cache; it persists only as long as the referent is more strongly referenced.

The reference will be cleared before it is added to its reference queue (and so the reference queue cannot resurrect the object).

`PhantomReference` provides a way to attach extra finalization to an object without actually using finalizers (which have several problems, including the ability to resurrect the object). The phantom reference *always* presents `null` as its value and is therefore itself useless as a reference. Phantom references are enqueued after the object is finalized and therefore at a point when there can be no references to the object within the VM at all. However, the object itself cannot be deallocated until the phantom references are all cleared or themselves deallocated, which I believe is for the convenience of native code that may hold a dangling weak reference to the referent (or which may be able to directly read the reference).

.NET

The `WeakReference` class in .NET is similar to Java's `WeakReference` class in that the value cannot be accessed directly; it must be accessed via the `Target` property, which may yield `null`. The reference may be reseated to a different value.

Weak references may be created *long*, which permits the target object to be finalized but not actually deallocated.

Python

A `weakref` acts like a function object; it is created with a particular value, which cannot be reseated. The function will yield `None` if the referent is collected.

There is library functionality to automatically proxy a value as a weak reference. An exception is thrown if an operation is performed on the proxy but the referent has been collected.

A `weakref` may be constructed with a callback function. The callback will be called after the weak reference is cleared; it is, however, passed the weak ref object itself.

Ruby

A `WeakRef` is automatically a proxy for an object. There is a `weakref_alive` method to query whether the reference is still alive; another other operation will cause an exception to be thrown.

Rust

As far as I can tell, there is nothing like a weak reference in Rust at the moment.

A *managed pointer* (`@int`) is a strong reference subject to GC.

An *owning pointer* (`~int`) is a strong reference that cannot be cloned (copying the pointer actually copies the underlying data).

A *borrowed pointer* (`&int`) is essentially a dangling weak reference that is subject to static restrictions which ensure that it doesn't actually dangle. It is thus primarily a performance optimization.

A *raw pointer* (`*int`) is a dangling weak reference.

Haskell

Yes, of course Haskell has weak references.

A `Weak t` is an association between a hidden key and a visible value of type `t`. `doRefWeak theRef` is an `IO (Maybe t)`.

A weak reference may be constructed with an optional `IO ()` which will be run when the referent is collected. This finalizer may (somehow) refer to the key and value without itself keeping them alive; it is also explicitly permitted to resurrect them.

Use Cases

There are many problems that are potentially addressable with functionality like weak references. It is not at all obvious that they should be addressed with the same language feature.

Back references

Given that Swift is not cycle-collecting, far and away the most important use case in the static reference graph is that of the *back-reference*: a reference R to an object which holds a strong reference (possibly indirectly) to the object holding R. Examples include:

- A 'previousNode' pointer in a doubly-linked list.
- A 'parent' pointer in a render tree.
- An edge in a general graph structure.

These have several properties in common:

- Using strong references would require a lot of explicit code to tear down the reference cycles.
- These references may be accessed very frequently, so performance is important.
- It is not always feasible to make these references valid immediately on construction.
- Traversing a reference after the referent is deallocated is likely a sign that something has been kept alive longer than it was meant to be. However, programmers may reasonably differ about the correct response to this: crashing, and therefore encouraging the programmer to track down a root

cause, or simply writing the operation to handle both cases correctly. Ultimately, this choice comes down to philosophy.

Caches

Weak caches are used in order to prevent a cache from taking over all available memory. By being tied to the reachability of a value, the cache prevents entries from spuriously expiring when their values are still in active use; but by using weak references, the cache permits the system to deallocate values that are no longer in use.

Generally, a data structure using weak references extensively also needs some way to receive notification that the weak reference was collected. This is because entries in the data structure are likely to have significant overhead even if the value is collected. A weak data structure which receives no notification that a reference has been invalidated must either allow these entries to accumulate indefinitely or must periodically scan the entire structure looking for stale entries.

A weak reference which permits immediate deallocation of its referent when the last strong reference is dropped is substantially less useful for the implementation of a weak cache. It is a common access pattern (for, say, a memoizing cache) for a value to be looked up many times in rapid succession, but for each use to be temporarily disjoint from the others. A naive use of weak references in this case will simply cause the cache to thrash. This problem is less likely to arise in an environment with nondeterministic collection because the entry is likely to service multiple lookups between collections.

It is likely that users implementing weak data structures would prefer a highly flexible infrastructure centered around resurrection and notifications of reaching a zero refcount than a more rigid system built directly into the language. Since the Swift model is built around statically-inserted operations rather than a memory scanner, this is much more workable.

External Finalization

Finalization models built around calling a method on the finalized object (such as Objective-C's `-dealloc`) suffer from a number of limitations and problems:

- Since the method receives a pointer to the object being deallocated, the implementation must guard against attempts to resurrect the object. This may complicate and/or slow down the system's basic reference-management logic, which tends to be quite important for performance.
- Since the method receives a pointer to the object being deallocated, the implementation must leave the object at least a minimally valid state until the user code is complete. For example, the instance variables of a subclass cannot be destroyed until a later phase of destruction, because a superclass finalizer might invoke subclass behavior. (This assumes that

the dynamic type of the object does not change during destruction, which is an alternative that brings its own problems.)
- Finalization code must be inherent to the object; other objects cannot request that code be run when the object is deallocated. For example, an object that registers itself to observe a certain event source must explicitly deregister itself in a finalizer; the event source cannot simply automatically drop the object when it is deallocated.

Optimization

Functions often create a large number of temporary references. In a reference-counting environment like Swift, these references require the implementation to implicitly perform operations to increment and decrement the reference count. These operations can be quite fast, but they are not free, and our experience has been that the accumulated cost can be quite significant. A straightforward local static analysis can eliminate many operations, but many others will be blocked by abstraction barriers, chiefly dynamically-dispatched calls. Therefore, if Swift is to allow precise performance control, it is important to be able to allow motivated users to selectively control the emission of reference-counting operations.

This sort of control necessarily permits the creation of dangling weak references and so is not safe.

Proposal Overview

Looking at these use-cases, there are two main thrusts:

- There is a general need to set up back references to objects. These references must be designed for convenient use by non-expert users.
- There are a number of more sophisticated use cases which require notification or interruption of deallocation; these can be used in the implementation of higher-level abstractions like weak caches. Here it is reasonable to expect more user expertise, such that power and flexibility should take priority over ease of use.

The second set of use cases should be addressed by library types working on top of basic runtime support.

The first set of use cases will require more direct language support. To that end, I propose adding two new variable attributes, `@weak` and `@unowned`. I also propose a small slate of new features which directly address the problem of capturing a value in a closure with a different strength of reference than it had in the enclosing context.

Proposed Variable Attributes

In the following discussion, a "variable-like" declaration is any declaration which binds a name to a (possibly mutable) value of arbitrary type. Currently this is just `var`, but this proposal also adds `capture`, and we may later add more variants, such as `const` or `val` or the like.

`@weak`

`weak` is an attribute which may be applied to any variable-like declaration of reference type `T`. For type-system purposes, the variable behaves like a normal variable of type `Optional<T>`, except:

- it does not maintain a +1 reference count invariant and
- loading from the variable after the current referent (if present) has started destruction will result in a `Nothing` value, indistinguishable from the normal case.

The semantics are quite similar to weak references in other environments (particularly Objective-C) except that the change in formal type forces the user of the value to check its validity before using it.

It doesn't really matter how the compiler would find the type `Optional<T>`; compiler-plus-stdlib magic, most likely. I do not think the type should be `Weak<T>` because that would effectively make this a type attribute and thus make it too easy to accidentally preserve the value as a weak reference. See the section discussing type attributes vs. declaration attributes.

Giving the variable a consistent type of `Optional<T>` permits the user to assign `Nothing` into it and therefore removes the somewhat odd expressive possibility of a reference that can only be missing if the object has been deallocated. I think this is an acceptable cost of making a cleaner feature.

One alternative to using `Optional<T>` would be to simply treat the load as a potentially-failing operation, subject to the (not yet precisely designed) language rules for error handling. This would require the runtime to potentially synthesize an error event, which could then propagate all the way to the end-user if the programmer accidentally failed to check the result in a context that permitted error propagation outwards. That's bad.

A slightly different alternative would be to treat it as a form of error orthogonal to the standard user-error propagation. This would be cleaner than changing the type of the variable but can't really be designed without first having a solid error-handling design.

`@unowned`

`unowned` is an attribute which may be applied to any "variable-like" declaration of reference type `T`. For type-system purposes, the variable behaves exactly like a normal variable of type `T`, except:

- it does not maintain a +1 reference count invariant and
- loading from the variable after the referent has started destruction causes an assertion failure.

This is a refinement of `weak` focused more narrowly on the case of a back reference with relatively tight validity invariants. This is also the case that's potentially optimizable to use dangling weak references; see below.

This name isn't really optimal. We've considered several different candidates:

- `weak` is a poor choice because our semantics are very different from weak references in other environments where it's valid to access a cleared reference. Plus, we need to expose those semantics, so the name is claimed.
- `backref` is strongly evocative of the major use case in the static reference graph; this would encourage users to use it for back references and to consider alternatives for other cases, both of which I like. The latter also makes the husk-leaking implementation (see below) more palatable. It also contrasts very well with `weak`. However, its evocativeness makes it unwieldy to use for local reference-counting optimizations.
- `dangling` is more general than `backref`, but it has such strong negative associations that it wouldn't be unreasonable for users to assume that it's unsafe (with all the pursuant debugging difficulties) based on the name alone. I don't think we want to discourage a feature that can help users build tighter invariants on their classes.
- `unowned` is somewhat cleaner-looking, and it isn't as tied to a specific use case, but it does not contrast with `weak` *at all*; only someone with considerable exposure to weak references would understand why we named each one the way we did, and even they are likely to roll their eyes at us. But it's okay for a working proposal.

Asserting and Uncheckable There should not be a way to check whether a `unowned` reference is still valid.

- An invalid back-reference is a consistency error that we should encourage programmers to fix rather than work around by spot-testing for validity.
- Contrariwise, a weak reference that might reasonably be invalidated during active use should be checked for validity at *every* use. We can provide a simple library facility for this pattern.
- Permitting implicit operations like loads to fail in a recoverable way may end up complicating the language model for error-handling.
- By disallowing recovery, we create a model where the only need to actually register the weak reference with the system is to enable a consistency check. Users who are confident in the correctness of their program may therefore simply disable the consistency check without affecting the semantics of the program. In this case, that leaves the variable a simple dangling-weak reference.

Implementation The standard implementation for a weak reference requires the address of the reference to be registered with the system so that it can be cleared when the referent is finalized. This has two problems:

- It forces the runtime to maintain a side-table mapping objects to the list of weak references; generally this adds an allocation per weakly-referenced object.
- It forces the representation of weak references to either be non-address-invariant or to introduce an extra level of indirection.

For some use cases, this may be warranted; for example, in a weak cache it might come out in the noise. But for a simple back-reference, these are substantial penalties.

Dave Z. has proposed instead using a weak refcount, analogous to a strong refcount. Ownership of a weak retain can be easily transferred between locations, and it does not require a side-table of an object's weak references. However, it does have a very important downside: since the system cannot clear all the references, it is impossible to actually deallocate an object that is still weakly-referenced (although it can be finalized). Instead, the system must wait for all the weak references to at least be accessed. We call this "husk leaking".

This downside could be a problem for a general weak reference. However, it's fine for a back-reference, which we expect to typically be short-lived after its referent is finalized.

Declaration Attribute or Type Attribute

This proposal describes `weak` and `unowned` as declaration attributes, not type attributes.

As declaration attributes, `@unowned` and `weak` would be permitted on any `var` declaration of reference type. Their special semantics would be a property only of the declaration; in particular, they would not change the type (beyond the shift to `Optional<T>` for `weak`) , and more generally the type-checker would not need to know about them. The implementation would simply use different behavior when loading or storing that variable.

As a type attribute, `weak` and `@unowned` would be permitted to appear at arbitrary nested locations in the type system, such as tuple elements, function result types, or generic arguments. It would be possible to have both lvalues and rvalues of so-qualified type, and the type checker would need to introduce implicit conversions in the right places.

These implicit conversions require some thought. Consider code like the following:

```
func moveToWindow(_ newWindow : Window) {
  var oldWindow = self.window   // an @unowned back reference
  oldWindow.hide()              // might remove the UI's strong reference
  oldWindow.remove(self)
```

```
  newWindow.add(self)
}
```

It would be very unfortunate if the back-reference nature of the `window` property were somehow inherited by `oldWindow`! Something, be it a general rule on loading back-references or a type-inference rule, must introduce an implicit conversion and cause the `unowned` qualifier to be stripped.

That rule, however, is problematic for generics. A key goal of generics is substitutability: the semantics of generic code should match the semantics of the code you'd get from copy-pasting the generic code and substituting the arguments wherever they're written. But if a generic argument can be `@unowned T`, then this won't be true; consider:

```
func foo<T>(x : T) {
  var y = x
  ...
}
```

In substituted code, `y` would have the qualifier stripped and become a strong reference. But the generic type-checker cannot statically recognize that this type is `unowned`-qualified, so in order to match semantics, this decision must be deferred until runtime, and the type-checker must track the unqualified variant of `T`. This adds a great deal of complexity, both to the implementation and to the user model, and removes many static optimization opportunities due to potential mismatches of types.

An alternative rule would be to apply an implicit "decay" to a strong reference only when a type is known to be a `unowned` type. It could be argued that breaking substitution is not a big deal because other language features, like overloading, can already break it. But an overlapping overload set with divergent behavior is a poor design which itself violates substitution, whereas this would be a major unexcused deviation. Furthermore, preserving the weakness of a reference is not a safe default, because it permits the object to be destroyed while a reference is still outstanding.

In addition, any implementation model which permits the safety checks on `unowned`s to be disabled will require all code to agree about whether or not the checks are enabled. When this information is tied to a declaration, this is easy, because declarations are ultimately owned by a particular component, and we can ask how that component is compiled. (And we can demand that external APIs commit to one level of safety or the other before publishing.) The setting for a type, on the other hand, would have to be determined by the module which "wrote the type", but again this introduces a great deal of complexity which one can only imagine settling on the head of some very confused user.

For all these reasons, I feel that making `weak` and `unowned` type attributes would be unworkable. However, there are still costs to making them declaration attributes:

- It forces users to use awkward workarounds if they want to make, say, arrays of back-references.
- It makes back-references less composable by, say, preventing them from being stored in a tuple.
- It complicates SIL and IR-gen by making the rules for manipulating a physical variable depend on more than just the type of the variable.
- It automatically enables certain things (like passing the address of a `@unowned` variable of type `T` to a `inout T` parameter) that perhaps ought to be more carefully considered.

The first two points can be partly compensated for by adding library types to wrap a back-reference. Accessing a wrapped reference will require extra syntax, and it runs the same risk of accidentally preserving the weakness of a reference that I discussed above. However, note that these problems are actually at odds: requiring extra syntax to access the wrapped reference will leave breadcrumbs making it clear when the change-over occurs. For more on this, see the library section.

`weak`-Capable Types

Swift reference types can naturally be made to support any kind of semantics, and I'm taking it on faith that we could enhance ObjC objects to support whatever extended semantics we want. There are, however, certain Swift value types which have reference-like semantics that it could be useful to extend `weak` (and/or `unowned`) to:

- Being able to conveniently form an optional back-reference seems like a core requirement. If `weak` were a type attribute, we could just expect users to write `Optional<@weak T>`; as a declaration attribute, this is substantially more difficult. I expect this to be common enough that it'll be unreasonable to ask users to use `Optional<WeakReference<T>>`.
- Being able to form a back-reference to a slice or a string seems substantially less important.

One complication with extending `weak` to value types is that generally the implementing type will need to be different from the underlying value type. Probably the best solution would be to hide the use of the implementing type from the type system outside of the wellformedness checks for the variable; SIL-gen would lower the field to its implementing type using the appropriate protocol conformances.

As long as we have convenient optional back-references, though, we can avoid designing a general feature for 1.0.

Generic Weak Support

All other uses for weak references can be glossed as desiring some amount of additional work to occur when the strong reference count for an object reaches

zero. This necessarily entails a global side-table of such operations, but I believe that's acceptable as long as it's relegated to less common use-cases.

It is important that the notification mechanism not require executing code re-entrantly during the finalization process.

I suggest adopting an interface centered around the Java concept of a `ReferenceQueue`. A reference structure is registered with the runtime for a particular object with a particular set of flags and an optional reference queue:

```
struct Reference {
  void *Referent; // must be non-null upon registration
  struct ReferenceQueue *Queue; // must be valid or null
  size_t Reserved[2];
};

void swift_registerReference(struct Reference *reference,
                             size_t flags);
```

The user/library code is responsible for allocating these structures and initializing the first two fields, and it may include arbitrary fields before or after the `Reference` section, but while the reference is registered with the runtime, the entire `Reference` section becomes reserved and user/library code must not access it in any way.

The flags include:

- A priority. Should be constrained to two or three bits. References are processed in order of decreasing priority; as long as a reference still exists with higher priority, references with lower priority cannot be processed. Furthermore, as long as any reference exists, the referent cannot be finalized.
- Whether to automatically clear the reference when processing it. Note that a cleared reference is still considered to be registered with the runtime.

These could be combined so that e.g. even priorities cause an automatic clear and odd priorities do not; this would avoid some odd effects.

The runtime may assume that explicit user operations on the same reference will not race with each other. However, user operations on different references to the same referent may be concurrent, either with each other or with other refcount operations on the referent.

The operations on references are as follows:

```
void *swift_readReference(struct Reference *reference);
```

This operation atomically either produces a strong reference to the referent of the given object or yields `null` if the referent has been finalized (or if the referent is `null`). The reference must currently be registered with the runtime:

```
void swift_writeReference(struct Reference *reference,
                          void *newReferent);
```

This operation changes the referent of the reference to a new object, potentially `null`. The argument is taken at +0. The reference must currently be registered with the runtime. The reference keeps the same flags and reference queue:

```
void swift_unregisterReference(struct Reference *Reference);
```

This operation clears a reference, removes it from its reference queue (if it is enqueued), and unregisters it from the runtime. The reference must currently be registered with the runtime.

I propose the following simple interface to a ReferenceQueue; arguably, however, it ought to be a reference-counted library type with a small amount of native implementation:

```
struct ReferenceQueue;
struct ReferenceQueue *swift_createReferenceQueue(void);
```

Allocate a new reference queue:

```
void swift_destroyReferenceQueue(struct ReferenceQueue *queue);
```

Destroy a reference queue. There must not be any references with this queue currently registered with the runtime:

```
struct Reference *swift_pollReferenceQueue(struct ReferenceQueue *queue);
```

Proposed Rules for Captures within Closures

When a variable from an enclosing context is referenced in a closure, by default it is captured by reference. Semantically, this means that the variable and the enclosing context(s) will see the variable as a single, independent entity; modifications will be seen in all places. In terms of the reference graph, each context holds a strong reference to the variable itself. (In practice, most local variables captured by reference will not require individual allocation; usually they will be allocated as part of the closure object. But in the formalism, they are independent objects.)

Closures therefore make it fairly easy to introduce reference cycles: for example, an instance method might install a closure as an event listener on a child object, and if that closure refers to `self`, a reference cycle will be formed. This is an indisputable drawback of an environment which cannot collect reference cycles.

Relatively few languages both support closures and use reference-counting. I'm not aware of any that attempt a language solution to the problem; the usual solution is to the change the captured variable to use weak-reference semantics, usually by copying the original into a new variable used only for this purpose. This is annoyingly verbose, clutters up the enclosing scope, and duplicates information across multiple variables. It's also error-prone: since both names are in scope, it's easy to accidentally refer to the wrong one, and explicit capture lists only help if you're willing to be very explicit.

A better alternative (which we should implement in Objective-C as well) is to permit closures to explicitly specify the semantics under which a variable is captured. In small closures, it makes sense to put this near the variable reference; in larger closures, this can become laborious and redundant, and a different mechanism is called for.

In the following discussion, a *var-or-member expression* is an expression which is semantically constrained to be one of:

- A reference to a local variable-like declaration from an enclosing context.
- A member access thereof, possibly recursively.

Such expressions have two useful traits:

- They always end in an identifier which on some level meaningfully identifies the object.
- Evaluating them is relatively likely (but not guaranteed) to not have interesting side effects, and so we feel less bad about apparently shifting their evaluation around.

Decorated Capture References

Small closures are just as likely to participate in a reference cycle, but they suffer much more from extraneous syntax because they're more likely to be center-embedded in interesting expressions. So if there's anything to optimize for in the grammar, it's this.

I propose this fairly obvious syntax:

```
button1.setAction { unowned(self).tapOut() }
button2.setAction { if (weak(self)) { weak(self).swapIn() } }
```

The operand is evaluated at the time of closure formation. Since these references can be a long way from the top of the closure, we don't want to allow a truly arbitrary expression here, because the order of side-effects in the surrounding context could be very confusing. So we require the operand to be an `expr-var-or-member`. More complicated expressions really ought to be hoisted out to a separate variable for legibility anyway.

I do believe that being able to capture the value of a property (particularly of `self`) is very important. In fact, it's important independent of weak references. It is often possible to avoid a reference cycle by simply capturing a specific property value instead of the base object. Capturing by value is also an expressivity improvement: the programmer can easily choose between working with the property's instantaneous value (at the time the closure is created) or its current value (at the time the closure is invoked).

Therefore I also suggest a closely-related form of decoration:

```
button3.setAction { capture(self.model).addProfitStep() }
```

This syntax would force `capture` to become a real keyword.

All of these kinds of decorated references are equivalent to adding a so-attributed `capture` declaration (see below) with a nonce identifier to the top of the closure:

```
button1.setAction {
  capture @unowned _V1 = self
  _V1.tapOut()
}
button2.setAction {
  capture @weak _V2 = self
  if (_V2) { _V2.swapIn() }
}
button3.setAction {
  capture _V3 = self.model
  _V3.addProfitStep()
}
```

If the operand of a decorated capture is a local variable, then that variable must not be the subject of an explicit `capture` declaration, and all references to that variable within the closure must be identically decorated.

The requirement to decorate all references can add redundancy, but only if the programmer insists on decorating references instead of adding an explicit `capture` declaration. Meanwhile, that redundancy aids both maintainers (by preventing refactors from accidentally removing the controlling decoration) and readers (who would otherwise need to search the entire closure to understand how the variable is captured).

Captures with identical forms (the same sequence of members of the same variable) are merged to the same capture declaration. This permits type refinement to work as expected, as seen above with the `weak` capture. It also guarantees the elimination of some redundant computation, such as with the `state` property in this example:

```
resetButton.setAction {
  log("resetting state to " + capture(self.state))
  capture(self.model).setState(capture(self.state))
}
```

I don't see any immediate need for other kinds of capture decoration. In particular, I think back references are likely to be the right kind of weak reference here for basically every use, and I don't think that making it easy to capture a value with, say, a zeroable weak reference is important. This is just an intuition deepened by hallway discussions and close examination of a great many test cases, so I concede it may prove to be misguided, in which case it should be easy enough to add a new kind of decoration (if we're willing to burn a keyword on it).

`capture` Declarations

This feature generalizes the above, removing some redundancy in large closures and adding a small amount of expressive power.

A `capture` declaration can only appear in a closure: an anonymous closure expression or `func` declaration that appears directly within a function. (By "directly" I mean not within, say, a local type declaration within the function). `capture` will need to at least become a context-sensitive keyword.

A closure may contain multiple `capture` declarations, but they must all appear at the very top. One reason is that they can affect the capture semantics within the closure, so someone reading the closure should be able to find them easily. Another reason is that they can involve executing interesting code in the enclosing context and so should reliably appear near the closure formation site in the source code:

```
decl                    ::= decl-capture
decl-capture            ::= 'capture' attribute-list '=' expr-var-or-member
decl-capture            ::= 'capture' attribute-list decl-capture-expr-list
decl-capture-expr-list  ::= expr-var-or-member
decl-capture-expr-list  ::= expr-var-or-member ',' decl-capture-expr-list
```

Both forms of `capture` declaration cause one or more fields to be created within the closure object. At the time of creating the closure, these fields are initialized with the result of evaluating an expression in the enclosing context. Since the expression is evaluated in the enclosing context, it cannot refer to "previous" captures; otherwise we could have some awkward ambiguities. I think we should reserve the right to not execute an initializer if the closure will never be called; this is more important for local `func` declarations than for anonymous closure expressions.

The fields introduced by `capture` declarations should be immutable by default, but programmers should be able to write something like `capture var ...` to make them mutable. Locking down on mutation isn't quite as important as it is with implicit captures (where it's easy to accidentally believe you're modifying the enclosing variable) or even explicit captures in C++11 lambdas (where copies of the lambda object will copy the capture field and thus produce mystifying behavior in uncareful code), but it's still a source of easy mistakes that should require manual intervention to enable. This all presumes that we eventually design mutability into the language, of course.

In the pattern-initializer form, the field names are given explicitly by the pattern. The abbreviated form borrows the name of the captured member or local variable. In either case, names should be subject to the usual shadowing rules, whatever they may be, with the exception that capturing an enclosing variable with the abbreviated form is not problematic.

Attributes on a `capture` declaration affect all the fields it declares.

Let's spell out some examples. I expect the dominant form to be a simple identifier:

```
capture @unowned foo
```

This captures the current value of whatever `foo` resolves to (potentially a member of `self`!) and binds it within the closure as a back-reference.

Permitting the slightly more general form:

```
capture window.title
```

allows users to conveniently capture specific values without mucking up the enclosing scope with tons of variables only needed for setting up the closure. In particular, this makes it easy to capture specific fields out of an enclosing `self` object instead of capturing the object itself; that, plus forcing uses of `self` to be explicit in closures, would help users to conveniently avoid a class of inadvertent retain cycles.

I've included the general pattern-initializer form mostly for ease of exposition. It adds no major expressivity improvements over just creating a variable in the enclosing context. It does avoid cluttering the enclosing scope with new variables and permits captures to be locally renamed, and it can very convenient if introducing a new variable in the enclosing scope would be complicated (for example, if there were a reason to prefer using a single statement there). I don't think it does any harm, but I wouldn't mourn it, either. I do think that generalizing the initializer to an arbitrary expression would be a serious mistake, because readers are naturally going to overlook code which occurs inside the closure, and promoting side effects there would be awful.

It would be nice to have a way to declare that a closure should not have any implicit captures. I don't have a proposal for that right now, but it's not important for weak references.

Nested Closures

It is important to spell out how these rules affect captures in nested closures.

Recall that all of the above rules can be transformed into `capture` declarations at the beginning of a closure, and that `capture` declarations always introduce a by-value capture instead of a by-reference capture.

A by-reference capture is always of either a local variable or a `capture` declaration. In neither case do we currently permit such captures to "drag in" other declarations silently, to the extent that this is detectable. This means that mutable `capture` declarations that are themselves captured by reference must be separately allocated from the closure object, much like what happens with normal locals captured by reference.

I've considered it quite a bit, and I think that a by-value capture of a variable from a non-immediately enclosing context must be made ill-formed. At the very

least, it must be ill-formed if either the original variable is mutable (or anything along the path is, if it involves properties) or the capture adds `@unowned`.

This rule will effectively force programmers to use extra variables or `capture`s as a way of promising validity of the internal captures.

The motivation for this prohibition is that the intent of such captures is actually quite ambiguous and/or dangerous. Consider the following code:

```
async { doSomething(); GUI.sync { unowned(view).fireCompleted() } }
```

The intent of this code is to have captured a back-reference to the value of `view`, but it could be to do so at either of two points in time. The language must choose, and in this hypothetical it must do so without further declaration of intent and without knowledge of when and how many times the closures will be called.

Suppose that we capture the value at the earlier point, when we form the outer closure. This will behave very surprisingly if `view` is in fact mutated; it may be easier to imagine this if, instead of a simple local variable, it was instead a path like `self.view`. And it's not clear that forming a back-reference at this earlier point is actually valid; it is easy to imagine code that would rely on the intermediate closure holding a strong reference to the view. Furthermore, and crucially, these issues are inextricable: we cannot somehow keep track of the mutable variable but only hold its value weakly.

But suppose instead that we capture the value at the later point. Then the intermediate closure will capture the `view` variable by reference, which means that in effect it will hold `view` strongly. This may actually completely subvert the user's desired behavior.

It's not clear to me right now whether this problem applies equally to explicit `capture` declarations. Somehow decorated expressions seem more ambiguous in intent, probably because the syntax is more thoughtlessly convenient. On the other hand, making the decoration syntax not just a shorthand for the explicit declarations makes the model more complex, and it may be over-complex already.

So in summary, it would be best to enforce a strong prohibition against these dangerous multi-level captures. We can tell users to introduce secondary variables when they need to do subtle things across several closure levels.

Library Directions

The library should definitely provide the following types:

- `UnownedReference<T>`: a fragile value type with a single public `unowned` field of type `T`. There should be an implicit conversion *from* `T` so that obvious initializations and assignments succeed. However, there should not be an implicit conversion *to* `T`; this would be dangerous because it

could hide bugs introduced by the way that e.g. naive copies into locals will preserve the weakness of the reference.

In keeping with our design for `unowned`, I think this type should actually be an alias to either `SafeUnownedReference<T>` or `UnsafeUnownedReference<T>` depending on the current component's build settings. The choice would be exported in binary modules, but for cleanliness we would also require public APIs to visibly commit to one choice or the other.

- `WeakReference<T>`: a fragile value type with a single public `weak` field of type `T`. As above, there should be an implicit conversion *from* `T` but no implicit conversion to `T` (or even `Optional<T>`). There is, however, no need for safe and unsafe variants.

The library should consider providing the following types:

- A simple, memory-sensitive weak cache.
- `Finalizer`: a value type which is constructed with a referent and a `() -> ()` function and which causes the function to be run (on a well-known dispatch queue?) when the object is finalized.