orphan:

The Swift Calling Convention

Contents

This whitepaper discusses the Swift calling convention, at least as we want it to be.

It's a basic assumption in this paper that Swift shouldn't make an implicit promise to exactly match the default platform calling convention. That is, if a C or Objective-C programmer manages to derive the address of a Swift function, we don't have to promise that an obvious translation of the type of that function will be correctly callable from C. For example, this wouldn't be guaranteed to work:

```
// In Swift:
func foo(_ x: Int, y: Double) -> MyClass { ... }

// In Objective-C:
extern id _TF4main3fooFTSiSd_CS_7MyClass(intptr_t x, double y);
```

We do sometimes need to be able to match C conventions, both to use them and to generate implementations of them, but that level of compatibility should be opt-in and site-specific. If Swift would benefit from internally using a better convention than C/Objective-C uses, and switching to that convention doesn't damage the dynamic abilities of our target platforms (debugging, dtrace, stack traces, unwinding, etc.), there should be nothing preventing us from doing so. (If we did want to guarantee compatibility on this level, this paper would be a lot shorter!)

Function call rules in high-level languages have three major components, each operating on a different abstraction level:

- the high-level semantics of the call (pass-by-reference vs. pass-by-value),
- the ownership and validity conventions about argument and result values ("+0" vs. "+1", etc.), and
- the "physical" representation conventions of how values are actually communicated between functions (in registers, on the stack, etc.).

We'll tackle each of these in turn, then conclude with a detailed discussion of function signature lowering.

High-level semantic conventions

The major division in argument passing conventions between languages is between pass-by-reference and pass-by-value languages. It's a distinction that only really makes sense in languages with the concept of an l-value, but Swift does, so it's pertinent.

In general, the terms "pass-by-X" and "call-by-X" are used interchangeably. It's unfortunate, because these conventions are argument specific, and functions can be passed multiple arguments that are each handled in a different way. As such, we'll prefer "pass-by-X" for consistency and to emphasize that these conventions are argument-specific.

Pass-by-reference

In pass-by-reference (also called pass-by-name or pass-by-address), if *A* is an l-value expression, *foo(A)* is passed some sort of opaque reference through which the original l-value can be modified. If *A* is not an l-value, the language may prohibit this, or (if pass-by-reference is the default convention) it may pass a temporary variable containing the result of *A*.

Don't confuse pass-by-reference with the concept of a *reference type*. A reference type is a type whose value is a reference to a different object; for example, a pointer type in C, or a class type in Java or Swift. A variable of reference type can be passed by value (copying the reference itself) or by reference (passing the variable itself, allowing it to be changed to refer to a different object). Note that references in C++ are a generalization of pass-by-reference, not really a reference type; in C++, a variable of reference type behaves completely unlike any other variable in the language.

Also, don't confuse pass-by-reference with the physical convention of passing an argument value indirectly. In pass-by-reference, what's logically being passed is a reference to a tangible, user-accessible object; changes to the original object will be visible in the reference, and changes to the reference will be reflected in the original object. In an indirect physical convention, the argument is still logically an independent value, no longer associated with the original object (if there was one).

If every object in the language is stored in addressable memory, pass-by-reference can be easily implemented by simply passing the address of the object. If an l-value can have more structure than just a single, independently-addressable object, more information may be required from the caller. For example, an array argument in FORTRAN can be a row or column vector from a matrix, and so arrays are generally passed as both an address and a stride. C and C++ do have unaddressable l-values because of bitfields, but they forbid passing bitfields by reference (in C++) or taking their address (in either language), which greatly simplifies pointer and reference types in those languages.

FORTRAN is the last remaining example of a language that defaults to pass-by-reference. Early FORTRAN implementations famously passed constants by passing the address of mutable global memory initialized to the constant; if the callee modified its parameter (illegal under the standard, but...), it literally changed the constant for future uses. FORTRAN now allows procedures to explicitly take arguments by value and explicitly declare that arguments must be l-values.

However, many languages do allow parameters to be explicitly marked as pass-by-reference. As mentioned for C++, sometimes only certain kinds of l-values are allowed.

Swift allows parameters to be marked as pass-by-reference with *inout*. Arbitrary l-values can be passed. The Swift convention is to always pass an address; if the parameter is not addressable, it must be materialized into a temporary and then written back. See the accessors proposal for more details about the high-level semantics of *inout* arguments.

Pass-by-value

In pass-by-value, if *A* is an l-value expression, *foo(A)* copies the current value there. Any modifications *foo* makes to its parameter are made to this copy, not to the original l-value.

Most modern languages are pass-by-value, with specific functions able to opt in to pass-by-reference semantics. This is exactly what Swift does.

There's not much room for variation in the high-level semantics of passing arguments by value; all the variation is in the ownership and physical conventions.

Ownership transfer conventions

Arguments and results that require cleanup, like an Objective-C object reference or a non-POD C++ object, raise two questions about responsibility: who is responsible for cleaning it up, and when?

These questions arise even when the cleanup is explicit in code. C's *strdup* function returns newly-allocated memory which the caller is responsible for freeing, but *strtok* does not. Objective-C has standard naming conventions that describe which functions return objects that the caller is responsible for releasing, and outside of ARC these must be followed manually. Of course, conventions designed to be implemented by programmers are often designed around the simplicity of that implementation, rather than necessarily being more efficient.

Pass-by-reference arguments

Pass-by-reference arguments generally don't involve a *transfer* of ownership. It's assumed that the caller will ensure that the referent is valid at the time of the call, and that the callee will ensure that the referent is still valid at the time of return.

FORTRAN does actually allow parameters to be tagged as out-parameters, where the caller doesn't guarantee the validity of the argument before the call. Objective-C has something similar, where an indirect method argument can be marked *out*; ARC takes advantage of this with autoreleasing parameters to avoid a copy into the writeback temporary. Neither of these are something we semantically care about supporting in Swift.

There is one other theoretically interesting convention question here: the argument has to be valid before the call and after the call, but does it have to valid during the call? Swift's answer to this is generally "yes". Swift does have *inout* aliasing rules that allow a certain amount of optimization, but the compiler is forbidden from exploiting these rules in any way that could cause memory corruption (at least in the absence of race conditions). So Swift has to ensure that an *inout* argument is valid whenever it does something (including calling an opaque function) that could potentially access the original l-value.

If Swift allowed local variables to be captured through *inout* parameters, and therefore needed to pass an implicit owner parameter along with an address, this owner parameter would behave like a pass-by-value argument and could use any of the conventions listed below. However, the optimal convention for this is obvious: it should be *guaranteed*, since captures are very unlikely and callers are almost always expected to use the value of an *inout* variable afterwards.

Pass-by-value arguments

All conventions for this have performance trade-offs.

We're only going to discuss *static* conventions, where the transfer is picked at compile time. It's possible to have a *dynamic* convention, where the caller passes a flag indicating whether it's okay to directly take responsibility for the value, and the callee can (conceptually) return a flag indicating whether it actually did take responsibility for it. If copying is extremely expensive, that can be worthwhile; otherwise, the code cost may overwhelm any other benefits.

This discussion will ignore one particular impact of these conventions on code size. If a function has many callers, conventions that require more code in the caller are worse, all else aside. If a single call site has many possible targets, conventions that require more code in the callee are worse, all else aside. It's not really reasonable to decide this in advance for unknown code; we could maybe make rules about code calling system APIs, except that system APIs are by definition locked down, and we can't change them. It's a reasonable thing to consider changing with PGO, though.

Responsibility

A common refrain in this performance analysis will be whether a function has responsibility for a value. A function has to get a value from *somewhere*:

- A caller is usually responsible for the return values it receives: the callee generated the value and the caller is responsible for destroying it. Any other convention has to rely on heavily restricting what kind of value can be returned. (If you're thinking about Objective-C autoreleased results, just accept this for now; we'll talk about that later.)
- A function isn't necessarily responsible for a value it loads from memory. Ignoring race conditions, the function may be able to immediately use the value without taking any specific action to keep it valid.
- A callee may or may not be responsible for a value passed as a parameter, depending on the convention it was passed with.
- A function might come from a source that doesn't necessarily make the function responsible, but if the function takes an action which invalidates the source before using the value, the function has to take action to keep the value valid. At that point, the function has responsibility for the value despite its original source.

  For example, a function *foo()* might load a reference *r* from a global variable *x*, call an unknown function *bar()*, and then use *r* in some way. If *bar()* can't possibly overwrite *x*, *foo()* doesn't have to do anything to keep *r* alive across the call; otherwise it does (e.g. by retaining it in a refcounted environment). This is a situation where humans are often much smarter than compilers. Of course, it's also a situation where humans are sometimes insufficiently conservative.

A function may also require responsibility for a value as part of its operation:

- Since a variable is always responsible for the current value it stores, a function which stores a value into memory must first gain responsibility for that value.
- A callee normally transfers responsibility for its return value to its caller; therefore it must gain responsibility for its return value before returning it.
- A caller may need to gain responsibility for a value before passing it as an argument, depending on the parameter's ownership-transfer convention.

Known conventions

There are three static parameter conventions for ownership worth considering here:

- The caller may transfer responsibility for the value to the callee. In SIL, we call this an **owned** parameter.

  This is optimal if the caller has responsibility for the value and doesn't need it after the call. This is an extremely common situation; for example, it comes up whenever a call result is immediately used as an argument. By giving the callee responsibility for the value, this convention allows the callee to use the value at a later point without taking any extra action to keep it alive.

  The flip side is that this convention requires a lot of extra work when a single value is used multiple times in the caller. For example, a value passed in every iteration of a loop will need to be copied/retained/whatever each time.
- The caller may provide the value without any responsibility on either side. In SIL, we call this an **unowned** parameter. The value is guaranteed to be valid at the moment of the call, and in the absence of race conditions, that guarantee can be assumed to continue unless the callee does something that might invalidate it. As discussed above, humans are often much smarter than computers about knowing when that's possible.

  This is optimal if the caller can acquire the value without responsibility and the callee doesn't require responsibility of it. In very simple code --- e.g., loading values from an array and passing them to a comparator function which just reads a few fields from each and returns --- this can be extremely efficient.

  Unfortunately, this convention is completely undermined if either side has to do anything that forces it to take action to keep the value alive. Also, if that happens on the caller side, the convention can keep values alive longer than is necessary. It's very easy for both sides of the convention to end up doing extra work because of this.

- The caller may assert responsibility for the value. In SIL, we call this a **guaranteed** parameter. The callee can rely on the value staying valid for the duration of the call.

  This is optimal if the caller needs to use the value after the call and either has responsibility for it or has a guarantee like this for it. Therefore, this convention is particularly nice when a value is likely to be forwarded by value a great deal.

  However, this convention does generally keep values alive longer than is necessary, since the outermost function which passed it as an argument will generally be forced to hold a reference for the duration. By the same mechanism, in refcounted systems, this convention tends to cause values to have multiple retains active at once; for example, if a copy-on-write array is created in one function, passed to another, stored in a mutable variable, and then modified, the callee will see a reference count of 2 and be forced to do a structural copy. This can occur even if the caller literally constructed the array for the sole and immediate purpose of passing it to the callee.

Analysis

Objective-C generally uses the unowned convention for object-pointer parameters. It is possible to mark a parameter as being consumed, which is basically the owned convention. As a special case, in ARC we assume that callers are responsible for keeping *self* values alive (including in blocks), which is effectively the *guaranteed* convention.

unowned causes a lot of problems without really solving any, in my experience looking at ARC-generated code and optimizer output. A human can take advantage of it, but the compiler is so frequently blocked. There are many common idioms (like chains of functions that just add default arguments at each step) have really awful performance because the compiler is adding retains and releases at every single level. It's just not a good convention to adopt by default. However, we might want to consider allowing specific function parameters to opt into it; sort comparators are a particularly interesting candidate for this. *unowned* is very similar to C++'s *const &* for things like that.

guaranteed is good for some things, but it causes a lot of silly code bloat when values are really only used in one place, which is quite common. The liveness / refcounting issues are also pretty problematic. But there is one example that's very nice for *guaranteed*: *self*. It's quite common for clients of a type to call multiple methods on a single value, or for methods to dispatch to multiple other methods, which are exactly the situations where *guaranteed* excels. And it's relatively uncommon (but not unimaginable) for a non-mutating method on a copy-on-write struct to suddenly store *self* aside and start mutating that copy.

owned is a good default for other parameters. It has some minor performance disadvantages (unnecessary retains if you have an unoptimizable call in a loop) and some minor code size benefits (in common straight-line code), but frankly, both of those points pale in importance to the ability to transfer copy-on-write structures around without spuriously increasing reference counts. It doesn't take too many unnecessary structural copies before any amount of reference-counting traffic (especially the Swift-native reference-counting used in copy-on-write structures) is basically irrelevant in comparison.

Result values

There's no major semantic split in result conventions like that between pass-by-reference and pass-by-value. In most languages, a function has to return a value (or nothing). There are languages like C++ where functions can return references, but that's inherently limited, because the reference has to refer to something that exists outside the function. If Swift ever adds a similar language mechanism, it'll have to be memory-safe and extremely opaque, and it'll be easy to just think of that as a kind of weird value result. So we'll just consider value results here.

Value results raise some of the same ownership-transfer questions as value arguments. There's one major limitation: just like a by-reference result, an actual *unowned* convention is inherently limited, because something else other than the result value must be keeping it valid. So that's off the table for Swift.

What Objective-C does is something more dynamic. Most APIs in Objective-C give you a very ephemeral guarantee about the validity of the result: it's valid now, but you shouldn't count on it being valid indefinitely later. This might be because the result is actually owned by some other object somewhere, or it might be because the result has been placed in the autorelease pool, a thread-local data structure which will (when explicitly drained by something up the call chain) eventually release that's been put into it. This autorelease pool can be a major source of spurious memory growth, and in classic manual reference-counting it was important to drain it fairly frequently. ARC's response to this convention was to add an optimization which attempts to prevent things from ending up in the autorelease pool; the net effect of this optimization is that ARC ends up with an owned reference regardless of whether the value was autoreleased. So in effect, from ARC's perspective, these APIs still return an owned reference, mediated through some extra runtime calls to undo the damage of the convention.

So there's really no compelling alternative to an owned return convention as the default in Swift.

Physical conventions

The lowest abstraction level for a calling convention is the actual "physical" rules for the call:

- where the caller should place argument values in registers and memory before the call,
- how the callee should pass back the return values in registers and/or memory after the call, and
- what invariants hold about registers and memory over the call.

In theory, all of these could be changed in the Swift ABI. In practice, it's best to avoid changes to the invariant rules, because those rules could complicate Swift-to-C interoperation:

- Assuming a higher stack alignment would require dynamic realignment whenever Swift code is called from C.
- Assuming a different set of callee-saved registers would require additional saves and restores when either Swift code calls C or is called from C, depending on the exact change. That would then inhibit some kinds of tail call.

So we will limit ourselves to considering the rules for allocating parameters and results to registers. Our platform C ABIs are usually quite good at this, and it's fair to ask why Swift shouldn't just use C's rules. There are three general answers:

- Platform C ABIs are specified in terms of the C type system, and the Swift type system allows things to be expressed which don't have direct analogues in C (for example, enums with payloads).
- The layout of structures in Swift does not necessarily match their layout in C, which means that the C rules don't necessarily cover all the cases in Swift.
- Swift places a larger emphasis on first-class structs than C does. C ABIs often fail to allocate even small structs to registers, or use inefficient registers for them, and we would like to be somewhat more aggressive than that.

Accordingly, the Swift ABI is defined largely in terms of lowering: a Swift function signature is translated to a C function signature with all the aggregate arguments and results eliminated (possibly by deciding to pass them indirectly). This lowering will be described in detail in the final section of this whitepaper.

However, there are some specific circumstances where we'd like to deviate from the platform ABI:

Aggregate results

As mentioned above, Swift puts a lot of focus on first-class value types. As part of this, it's very valuable to be able to return common value types fully in registers instead of indirectly. The magic number here is three: it's very common for copy-on-write value types to want about three pointers' worth of data, because that's just enough for some sort of owner pointer plus a begin/end pair.

Unfortunately, many common C ABIs fall slightly short of that. Even those ABIs that do allow small structs to be returned in registers tend to only allow two pointers' worth. So in general, Swift would benefit from a very slightly-tweaked calling convention that allocates one or two more registers to the result.

Implicit parameters

There are several language features in Swift which require implicit parameters:

Closures

Swift's function types are "thick" by default, meaning that a function value carries an optional context object which is implicitly passed to the function when it is called. This context object is reference-counted, and it should be passed *guaranteed* for straightforward reasons:

- It's not uncommon for closures to be called many times, in which case an *owned* convention would be unnecessarily expensive.
- While it's easy to imagine a closure which would want to take responsibility for its captured values, giving it responsibility for a retain of the context object doesn't generally allow that. The closure would only be able to take ownership of the captured values if it had responsibility for a *unique* reference to the context. So the closure would have to be written to do different things based on the uniqueness of the reference, and it would have to be able to tear down and deallocate the context object after stealing values from it. The optimization just isn't worth it.
- It's usually straightforward for the caller to guarantee the validity of the context reference; worst case, a single extra Swift-native retain/release is pretty cheap. Meanwhile, not having that guarantee would force many closure functions to retain their contexts, since many closures do multiple things with values from the context object. So *unowned* would not be a good convention.

Many functions don't actually need a context, however; they are naturally "thin". It would be best if it were possible to construct a thick function directly from a thin function without having to introduce a thunk just to move parameters around the missing context parameter. In the worst case, a thunk would actually require the allocation of a context object just to store the original function pointer; but that's only necessary when converting from a completely opaque function value. When the source function is known statically, which is far more likely, the thunk can just be a global function which immediately calls the target with the correctly shuffled arguments. Still, it'd be better to be able to avoid creating such thunks entirely.

In order to reliably avoid creating thunks, it must be possible for code invoking an opaque thick function to pass the context pointer in a way that can be safely and implicitly ignored if the function happens to actually be thin. There are two ways to achieve this:

- The context can be passed as the final parameter. In most C calling conventions, extra arguments can be safely ignored; this is because most C calling conventions support variadic arguments, and such conventions inherently can't rely on the callee knowing the extent of the arguments.

  However, this is sub-optimal because the context is often used repeatedly in a closure, especially at the beginning, and putting it at the end of the argument list makes it more likely to be passed on the stack.
- The context can be passed in a register outside of the normal argument sequence. Some ABIs actually even reserve a register for this purpose; for example, on x86-64 it's *%r10*. Neither of the ARM ABIs do, however.

Having an out-of-band register would be the best solution.

(Surprisingly, the ownership transfer convention for the context doesn't actually matter here. You might think that an *owned* convention would be prohibited, since the callee would fail to release the context and would therefore leak it. However, a thin function should always have a *nil* context, so this would be harmless.)

Either solution works acceptably with curried partial application, since the inner parameters can be left in place while transforming the context into the outer parameters. However, an *owned* convention would either prevent the uncurrying forwarder from tail-calling the main function or force all the arguments to be spilled. Neither is really acceptable; one more argument against an *owned* convention. (This is another example where *guaranteed* works quite nicely, since the guarantees are straightforward to extend to the main function.)

self

Methods (both static and instance) require a *self* parameter. In all of these cases, it's reasonable to expect that *self* will used frequently, so it's best to pass it in a register. Also, many methods call other methods on the same object, so it's also best if the register storing *self* is stable across different method signatures.

In static methods on value types, *self* doesn't require any dynamic information: there's only one value of the metatype, and there's usually no point in passing it.

In static methods on class types, *self* is a reference to the class metadata, a single pointer. This is necessary because it could actually be the class object of a subclass.

In instance methods on class types, *self* is a reference to the instance, again a single pointer.

In mutating instance methods on value types, *self* is the address of an object.

In non-mutating instance methods on value types, *self* is a value; it may require multiple registers, or none, or it may need to be passed indirectly.

All of these cases except mutating instance methods on value types can be partially applied to create a function closure whose type is the formal type of the method. That is, if class *A* has a method declared *func foo(_ x: Int) -> Double*, then *A.foo* yields a function of type *(Int) -> Double*. Assuming that we continue to feel that this is a useful language feature, it's worth considered how we could support it efficiently. The expenses associated with a partial application are (1) the allocation of a context object and (2) needing to introduce a thunk to forward to the original function. All else aside, we can avoid the allocation if the representation of *self* is compatible with the representation of a context object reference; this is essentially true only if *self* is a class instance using Swift reference counting. Avoiding the thunk is possible only if we successfully avoided the allocation (since otherwise a thunk is required in order to extract the correct *self* value from the allocated context object) and *self* is passed in exactly the same manner as a closure context would be.

It's unclear whether making this more efficient would really be worthwhile on its own, but if we do support an out-of-band context parameter, taking advantage of it for methods is essentially trivial.

Error handling

The calling convention implications of Swift's error handling design aren't yet settled. It may involve extra parameters; it may involve extra return values. Considerations:

- Callers will generally need to immediately check for an error. Being able to quickly check a register would be extremely convenient.
- If the error is returned as a component of the result value, it shouldn't be physically combined with the normal result. If the normal result is returned in registers, it would be unfortunate to have to do complicated logic to test for error. If the normal result is returned indirectly, contorting the indirect result with the error would likely prevent the caller from evaluating the call in-place.
- It would be very convenient to be able to trivially turn a function which can't produce an error into a function which can. This is an operation that we expect higher-order code to have do frequently, if it isn't completely inlined away. For example:

  ```
  // foo() expects its argument to follow the conventions of a
  // function that's capable of throwing.
  func foo(_ fn: () throws -> ()) throwsIf(fn)

  // Here we're passing foo() a function that can't throw; this is
  // allowed by the subtyping rules of the language.  We'd like to be
  // able to do this without having to introduce a thunk that maps
  // between the conventions.
  func bar(_ fn: () -> ()) {
    foo(fn)
  }
  ```

We'll consider two ways to satisfy this.

The first is to pass a pointer argument that doesn't interfere with the normal argument sequence. The caller would initialize the memory to a zero value. If the callee is a throwing function, it would be expected to write the error value into this argument; otherwise, it would naturally ignore it. Of course, the caller then has to load from memory to see whether there's an error. This would also either consume yet another register not in the normal argument sequence or have to be placed at the end of the argument list, making it more likely to be passed on the stack.

The second is basically the same idea, but using a register that's otherwise callee-save. The caller would initialize the register to a zero value. A throwing function would write the error into it; a non-throwing function would consider it callee-save and naturally preserve it. It would then be extremely easy to check it for an error. Of course, this would take away a callee-save register in the caller when calling throwing functions. Also, if the caller itself isn't throwing, it would have to save and restore that register.

Both solutions would allow tail calls, and the zero store could be eliminated for direct calls to known functions that can throw. The second is the clearly superior solution, but definitely requires more work in the backend.

Default argument generators

By default, Swift is resilient about default arguments and treats them as essentially one part of the implementation of the function. This means that, in general, a caller using a default argument must call a function to emit the argument, instead of simply inlining that emission directly into the call.

These default argument generation functions are unlike any other because they have very precise information about how their result will be used: it will be placed into a specific position in specific argument list. The only reason the caller would ever want to do anything else with the result is if it needs to spill the value before emitting the call.

Therefore, in principle, it would be really nice if it were possible to tell these functions to return in a very specific way, e.g. to return two values in the second and third argument registers, or to return a value at a specific location relative to the stack pointer (although this might be excessively constraining; it would be reasonable to simply opt into an indirect return instead). The function should also preserve earlier argument registers (although this could be tricky if the default argument generator is in a generic context and therefore needs to be passed type-argument information).

This enhancement is very easy to postpone because it doesn't affect any basic language mechanics. The generators are always called directly, and they're inherently attached to a declaration, so it's quite easy to take any particular generator and compatibly enhance it with a better convention.

ARM32

Most of the platforms we support have pretty good C calling conventions. The exceptions are i386 (for the iOS simulator) and ARM32 (for iOS). We really, really don't care about i386, but iOS on ARM32 is still an important platform. Switching to a better physical calling convention (only for calls from Swift to Swift, of course) would be a major improvement.

It would be great if this were as simple as flipping a switch, but unfortunately the obvious convention to switch to (AAPCS-VFP) has a slightly different set of callee-save registers: iOS treats *r9* as a scratch register. So we'd really want a variant of AAPCS-VFP that did the same. We'd also need to make sure that SJ/LJ exceptions weren't disturbed by this calling convention; we aren't really *supporting* exception propagation through Swift frames, but completely breaking propagation would be unfortunate, and we may need to be able to *catch* exceptions.

So this would also require some amount of additional support from the backend.

Function signature lowering

Function signatures in Swift are lowered in two phases.

Semantic lowering

The first phase is a high-level semantic lowering, which does a number of things:

- It determines a high-level calling convention: specifically, whether the function must match the C calling convention or the Swift calling convention.
- It decides the types of the parameters:
    - Functions exported for the purposes of C or Objective-C may need to use bridged types rather than Swift's native types. For example, a function that formally returns Swift's *String* type may be bridged to return an *NSString* reference instead.
    - Functions which are values, not simply immediately called, may need their types lowered to follow to match a specific generic abstraction pattern. This applies to functions that are parameters or results of the outer function signature.
- It identifies specific arguments and results which *must* be passed indirectly:
    - Some types are inherently address-only:
        - The address of a weak reference must be registered with the runtime at all times; therefore, any *struct* with a weak field must always be passed indirectly.
        - An existential type (if not class-bounded) may contain an inherently address-only value, or its layout may be sensitive to its current address.
        - A value type containing an inherently address-only type as a field or case payload becomes itself inherently address-only.
    - Some types must be treated as address-only because their layout is not known statically:
        - The layout of a resilient value type may change in a later release; the type may even become inherently address-

    only by adding a weak reference.
    - In a generic context, the layout of a type may be dependent on a type parameter. The type parameter might even be inherently address-only at runtime.
    - A value type containing a type whose layout isn't known statically itself generally will not have a layout that can be known statically.
  - Other types must be passed or returned indirectly because the function type uses an abstraction pattern that requires it. For example, a generic *map* function expects a function that takes a *T* and returns a *U*; the generic implementation of *map* will expect these values to be passed indirectly because their layout isn't statically known. Therefore, the signature of a function intended to be passed as this argument must pass them indirectly, even if they are actually known statically to be non-address-only types like (e.g.) *Int* and *Float*.

- It expands tuples in the parameter and result types. This is done at this level both because it is affected by abstraction patterns and because different tuple elements may use different ownership conventions. (This is most likely for imported APIs, where it's the tuple elements that correspond to specific C or Objective-C parameters.)

  This completely eliminates top-level tuple types from the function signature except when they are a target of abstraction and thus are passed indirectly. (A function with type *(Float, Int) -> Float* can be abstracted as *(T) -> U*, where *T == (Float, Int)*.)

- It determines ownership conventions for all parameters and results.

After this phase, a function type consists of an abstract calling convention, a list of parameters, and a list of results. A parameter is a type, a flag for indirectness, and an ownership convention. A result is a type, a flag for indirectness, and an ownership convention. (Results need ownership conventions only for non-Swift calling conventions.) Types will not be tuples unless they are indirect.

Semantic lowering may also need to mark certain parameters and results as special, for the purposes of the special-case physical treatments of *self*, closure contexts, and error results.

Physical lowering

The second phase of lowering translates a function type produced by semantic lowering into a C function signature. If the function involves a parameter or result with special physical treatment, physical lowering initially ignores this value, then adds in the special treatment as agreed upon with the backend.

General expansion algorithm

Central to the operation of the physical-lowering algorithm is the **generic expansion algorithm**. This algorithm turns any non-address-only Swift type in a sequence of zero or more **legal type**, where a legal type is either:

- an integer type, with a power-of-two size no larger than the maximum integer size supported by C on the target,
- a floating-point type supported by the target, or
- a vector type supported by the target.

Obviously, this is target-specific. The target also specifies a maximum voluntary integer size. The legal type sequence only contains vector types or integer types larger than the maximum voluntary size when the type was explicit in the input.

Pointers are represented as integers in the legal type sequence. We assume there's never a reason to differentiate them in the ABI as long as the effect of address spaces on pointer size is taken into account. If that's not true, this algorithm should be adjusted.

The result of the algorithm also associates each legal type with an offset. This information is sufficient to reconstruct an object in memory from a series of values and vice-versa.

The algorithm proceeds in two steps.

Typed layouts

First, the type is recursively analyzed to produce a **typed layout**. A typed layout associates ranges of bytes with either (1) a legal type (whose storage size must match the size of the associated byte range), (2) the special type **opaque**, or (3) the special type **empty**. Adjacent ranges mapped to **opaque** or **empty** can be combined.

For most of the types in Swift, this process is obvious: they either correspond to an obvious legal type (e.g. thick metatypes are pointer-sized integers), or to an obvious sequence of scalars (e.g. class existentials are a sequence of pointer-sized integers). Only a few cases remain:

- Integer types that are not legal types should be mapped as opaque.

- Vector types that are not legal types should be broken into smaller vectors, if their size is an even multiple of a legal vector type, or else broken into their components. (This rule may need some tinkering.)

- Tuples and structs are mapped by merging the typed layouts of the fields, as padded out to the extents of the aggregate with empty-mapped ranges. Note that, if fields do not overlap, this is equivalent to concatenating the typed layouts of the fields, in address order, mapping internal padding to empty. Bit-fields should map the bits they occupy to opaque.

  For example, given the following struct type:

  ```
  struct FlaggedPair {
    var flag: Bool
  ```

```
    var pair: (MyClass, Float)
  }
```

If Swift performs naive, C-like layout of this structure, and this is a 64-bit platform, typed layout is mapped as follows:

```
FlaggedPair.flag := [0: i1,                            ]
FlaggedPair.pair := [       8-15: i64, 16-19: float]
FlaggedPair      := [0: i1, 8-15: i64, 16-19: float]
```

If Swift instead allocates *flag* into the spare (little-endian) low bits of *pair.0*, the typed layout map would be:

```
FlaggedPair.flag := [0: i1                    ]
FlaggedPair.pair := [0-7: i64,    8-11: float]
FlaggedPair      := [0-7: opaque, 8-11: float]
```

- Unions (imported from C) are mapped by merging the typed layouts of the fields, as padded out to the extents of the aggregate with empty-mapped ranges. This will often result in a fully-opaque mapping.
- Enums are mapped by merging the typed layouts of the cases, as padded out to the extents of the aggregate with empty-mapped ranges. A case's typed layout consists of the typed layout of the case's directly-stored payload (if any), merged with the typed layout for its discriminator. We assume that checking for a discriminator involves a series of comparisons of bits extracted from non-overlapping ranges of the value; the typed layout of a discriminator maps all these bits to opaque and the rest to empty.

  For example, given the following enum type:

  ```
  enum Sum {
    case Yes(MyClass)
    case No(Float)
    case Maybe
  }
  ```

  If Swift, in its infinite wisdom, decided to lay this out sequentially, and to use invalid pointer values the class to indicate that the other cases are present, the layout would look as follows:

  ```
  Sum.Yes.payload        := [0-7: i64                 ]
  Sum.Yes.discriminator  := [0-7: opaque              ]
  Sum.Yes                := [0-7: opaque              ]
  Sum.No.payload         := [             8-11: float]
  Sum.No.discriminator   := [0-7: opaque              ]
  Sum.No                 := [0-7: opaque, 8-11: float]
  Sum.Maybe              := [0-7: opaque              ]
  Sum                    := [0-7: opaque, 8-11: float]
  ```

  If Swift instead chose to just use a discriminator byte, the layout would look as follows:

  ```
  Sum.Yes.payload        := [0-7: i64             ]
  Sum.Yes.discriminator  := [            8: opaque]
  Sum.Yes                := [0-7: i64,   8: opaque]
  Sum.No.payload         := [0-3: float           ]
  Sum.No.discriminator   := [            8: opaque]
  Sum.No                 := [0-3: float, 8: opaque]
  Sum.Maybe              := [            8: opaque]
  Sum                    := [0-8: opaque          ]
  ```

  If Swift chose to use spare low (little-endian) bits in the class pointer, and to offset the float to make this possible, the layout would look as follows:

  ```
  Sum.Yes.payload        := [0-7: i64             ]
  Sum.Yes.discriminator  := [0: opaque            ]
  Sum.Yes                := [0-7: opaque          ]
  Sum.No.payload         := [           4-7: float]
  Sum.No.discriminator   := [0: opaque            ]
  Sum.No                 := [0: opaque, 4-7: float]
  Sum.Maybe              := [0: opaque            ]
  Sum                    := [0-7: opaque          ]
  ```

The merge algorithm for typed layouts is as follows. Consider two typed layouts *L* and *R*. A range from *L* is said to *conflict* with a range from *R* if they intersect and they are mapped as different non-empty types. If two ranges conflict, and either range is mapped to a vector, replace it with mapped ranges for the vector elements. If two ranges conflict, and neither range is mapped to a vector, map them both to opaque, combining them with adjacent opaque ranges as necessary. If a range is mapped to a non-empty type, and the bytes in the range are all mapped as empty in the other map, add that range-mapping to the other map. *L* and *R* should now match perfectly; this is the result of the merge. Note that this algorithm is both associative and commutative.

Forming a legal type sequence

Once the typed layout is constructed, it can be turned into a legal type sequence.

Note that this transformation is sensitive to the offsets of ranges in the complete type. It's possible that the simplifications described here could be integrated directly into the construction of the typed layout without changing the results, but that's not yet proven.

In all of these examples, the maximum voluntary integer size is 4 (*i32*) unless otherwise specified.

If any range is mapped as a non-empty, non-opaque type, but its start offset is not a multiple of its natural alignment, remap it as opaque. For these purposes, the natural alignment of an integer type is the minimum of its size and the maximum voluntary integer size; the natural alignment of any other type is its C ABI type. Combine adjacent opaque ranges.

For example:

```
[1-2: i16, 4: i8, 6-7: i16]  ==>  [1-2: opaque, 4: i8, 6-7: i16]
```

If any range is mapped as an integer type that is not larger than the maximum voluntary size, remap it as opaque. Combine adjacent opaque ranges.

For example:

```
[1-2: opaque, 4: i8, 6-7: i16]  ==>  [1-2: opaque, 4: opaque, 6-7: opaque]
[0-3: i32, 4-11: i64, 12-13: i16]  ==>  [0-3: opaque, 4-11: i64, 12-13: opaque]
```

An *aligned storage unit* is an N-byte-aligned range of N bytes, where N is a power of 2 no greater than the maximum voluntary integer size. A *maximal* aligned storage unit has a size equal to the maximum voluntary integer size.

Note that any remaining ranges mapped as integers must fully occupy multiple maximal aligned storage units.

Split all opaque ranges at the boundaries of maximal aligned storage units. From this point on, never combine adjacent opaque ranges across these boundaries.

For example:

```
[1-6: opaque]  ==> [1-3: opaque, 4-6: opaque]
```

Within each maximal aligned storage unit, find the smallest aligned storage unit which contains all the opaque ranges. Replace the first opaque range in the maximal aligned storage unit with a mapping from that aligned storage unit to an integer of the aligned storage unit's size. Remove any other opaque ranges in the maximal aligned storage unit. Note that this can create overlapping ranges in some cases. For the purposes of this calculation, the last maximal aligned storage unit should be considered "full", as if the type had an infinite amount of empty tail-padding.

For example:

```
[1-2: opaque]  ==>  [0-3: i32]
[0-1: opaque]  ==>  [0-1: i16]
[0: opaque, 2: opaque]  ==>  [0-3: i32]
[0-9: fp80, 10: opaque]  ==>  [0-9: fp80, 10: i8]

// If maximum voluntary size is 8 (i64):
[0-9: fp80, 11: opaque, 13: opaque]  ==>  [0-9: fp80, 8-15: i64]
```

(This assumes that *fp80* is a legal type for illustrative purposes. It would probably be a better policy for the actual x86-64 target to consider it illegal and treat it as opaque from the start, at least when lowering for the Swift calling convention; for C, it is important to produce an *fp80* mapping for ABI interoperation with C functions that take or return *long double* by value.)

The final legal type sequence is the sequence of types for the non-empty ranges in the map. The associated offset for each type is the offset of the start of the corresponding range.

Only the final step can introduce overlapping ranges, and this is only possible if there's a non-integer legal type which:

- has a natural alignment less than half of the size of the maximum voluntary integer size or
- has a store size is not a multiple of half the size of the maximum voluntary integer size.

On our supported platforms, these conditions are only true on x86-64, and only of *long double*.

Deconstruction and Reconstruction

Given the address of an object and a legal type sequence for its type, it's straightforward to load a valid sequence or store the sequence back into memory. For the most part, it's sufficient to simply load or store each value at its appropriate offset. There are two subtleties:

- If the legal type sequence had any overlapping ranges, the integer values should be stored first to prevent overwriting parts of the other values they overlap.
- Care must be taken with the final values in the sequence; integer values may extend slightly beyond the ordinary storage size of the argument type. This is usually easy to compensate for.

The value sequence essentially has the same semantics that the value in memory would have: any bits that aren't part of the actual representation of the original type have a completely unspecified value.

Forming a C function signature

As mentioned before, in principle the process of physical lowering turns a semantically-lowered Swift function type (in implementation terms, a SILFunctionType) into a C function signature, which can then be lowered according to the usual rules for the ABI. This is, in fact, what we do when trying to match a C calling convention. However, for the native Swift calling convention, because we actively want to use more aggressive rules for results, we instead build an LLVM function type directly. We first construct a direct result type that we're certain the backend knows how to interpret according to our more aggressive desired rules, and then we use the expansion algorithm to construct a parameter sequence consisting solely of types with obvious ABI lowering that the backend can reliably handle. This bypasses the need to consult Clang for our own native calling convention.

We have this generic expansion algorithm, but it's important to understand that the physical lowering process does not just naively use the results of this algorithm. The expansion algorithm will happily expand an arbitrary structure; if that structure is very large, the algorithm might turn it into hundreds of values. It would be foolish to pass it as an argument that way; it would use up all the argument registers and basically turn into a very inefficient memcpy, and if the caller wanted it all in one place, they'd have to very painstakingly reassemble. It's much better to pass large structures indirectly. And with result values, we really just don't have a choice; there's only so many registers you can use before you have to give up and return indirectly. Therefore, even in the Swift native convention, the expansion algorithm is basically used as a first pass. A second pass then decides whether the expanded sequence is actually reasonable to pass directly.

Recall that one aspect of the semantically-lowered Swift function type is whether we should be matching the C calling convention or not. The following algorithm here assumes that the importer and semantic lowering have conspired in a very particular way to make that possible. Specifically, we assume is that an imported C function type, lowered semantically by Swift, will follow some simple structural rules:

- If there was a by-value *struct* or *union* parameter or result in the imported C type, it will correspond to a by-value direct parameter or return type in Swift, and the Swift type will be a nominal type whose declaration links back to the original C declaration.
- Any other parameter or result will be transformed by the importer and semantic lowering to a type that the generic expansion algorithm will expand to a single legal type whose representation is ABI-compatible with the original parameter. For example, an imported pointer type will eventually expand to an integer of pointer size.
- There will be at most one result in the lowered Swift type, and it will be direct.

Given this, we go about lowering the function type as follows. Recall that, when matching the C calling convention, we're building a C function type; but that when matching the Swift native calling convention, we're building an LLVM function type directly.

Results

The first step is to consider the results of the function.

There's a different set of rules here when we're matching the C calling convention. If there's a single direct result type, and it's a nominal type imported from Clang, then the result type of the C function type is that imported Clang type. Otherwise, concatenate the legal type sequences from the direct results. If this yields an empty sequence, the result type is *void*. If it yields a single legal type, the result type is the corresponding Clang type. No other could actually have come from an imported C declaration, so we don't have any real compatibility requirements; for the convenience of interoperation, this is handled by constructing a new C struct which contains the corresponding Clang types for the legal type sequence as its fields.

Otherwise, we are matching the Swift calling convention. Concatenate the legal type sequences from all the direct results. If target-specific logic decides that this is an acceptable collection to return directly, construct the appropriate IR result type to convince the backend to handle it. Otherwise, use the *void* IR result type and return the "direct" results indirectly by passing the address of a tuple combining the original direct results (*not* the types from the legal type sequence).

Finally, any indirect results from the semantically-lowered function type are simply added as pointer parameters.

Parameters

After all the results are collected, it's time to collect the parameters. This is done one at the time, from left to right, adding parameters to our physically-lowered type.

If semantic lowering has decided that we have to pass the parameter indirectly, we simply add a pointer to the type. This covers both mandatory-indirect pass-by-value parameters and pass-by-reference parameters. The latter can arise even in C and Objective-C.

Otherwise, the rules are somewhat different if we're matching the C calling convention. If the parameter is a nominal type imported from Clang, then we just add the imported Clang type to the Clang function type as a parameter. Otherwise, we derive the legal type sequence for the parameter type. Again, we should only have compatibility requirements if the legal type sequence has a single element, but for the convenience of interoperation, we collect the corresponding Clang types for all of the elements of the sequence.

Finally, if we're matching the Swift calling convention, derive the legal type sequence. If the result appears to be a reasonably small and efficient set of parameters, add their corresponding IR types to the function type we're building; otherwise, ignore the legal type sequence and pass the address of the original type indirectly.

Considerations for whether a legal type sequence is reasonable to pass directly:

- There probably ought to be a maximum size. Unless it's a single 256-bit vector, it's hard to imagine wanting to pass more than, say, 32 bytes of data as individual values. The callee may decide that it needs to reconstruct the value for some reason, and the larger the type gets, the more expensive this is. It may also be reasonable for this cap to be lower on 32-bit targets, but that might be dealt with better by the next restriction.
- There should also be a cap on the number of values. A 32-byte limit might be reasonable for passing 4 doubles. It's probably not reasonable for passing 8 pointers. That many values will exhaust all the parameter registers for just a single value. 4 is

probably a reasonable cap here.
- There's no reason to require the data to be homogeneous. If a struct contains three floats and a pointer, why force it to be passed in memory?

When all of the parameters have been processed in this manner, the function type is complete.