orphan:

Optimizer Effects: Summarizing and specifying function side effects

Contents

> **Note**
>
> This is a working document. Once we agree on the approach and terminology, this can move into docs/FunctionEffects.rst, and the codebase can be cleanup up a bit to reflect the consistent terminology.

Introduction

This document formalizes the effects that functions have on program state for the purpose of facilitating compiler optimization. By modeling more precise function effects, the optimizer can make more assumptions leading to more aggressive transformation of the program.

Function effects may be deduced by the compiler during program analysis. However, in certain situations it is helpful to directly communicate function effects to the compiler via function attributes or types. These source level annotations may or may not be statically enforceable.

This document specifies a comprehensive set of primitives and their semantics. These primitives are the optimizer's interface to function effects. They should be sufficient to express any analysis or annotation supported by the compiler that express a function's effect on program state.

Within the optimizer, SILEffectsAnalysis deduces function effects through analysis of SIL code. It's result directly maps to effects primitives.

Optimization of copy-on-Write data structures, such as Array, requires guaranteed function effects that cannot be deduced through analysis of SIL code. This necessitates a language for annotating functions. Some annotations are obviously specific to CoW semantics and cannot be defined in terms of general effects primitives: make_unique, preserve_unique, and projects_subobject. However, all other annotations required for CoW optimization are really guarantees regarding the program state that is affected by a CoW methods. These annotations should map precisely to a set of effects primitives.

For the sake of discussion, we use of Swift-level syntax for specifying effects primitives. It may be debatable whether we actually want to expose this syntax, but as explained above, some syntax will need to be exposed to build optimizable CoW types.

> **Note**
>
> [Andy] There is a lot of subtlety involved in specifying or summarizing function effects. I want to first put forth an underlying model for reasoning about the effects' semantics, demonstrate that we can prove soundness in all the cases we care about for CoW and any other foreseeable purpose, then go back if needed and add sugary syntax and proper defaults for specifying the effects. I won't get hung up on the attributes being too fine grained or tricky to use.

Effects Primitives

For the purpose of function effects, we identify program state that is known reachable via an argument, versus some other unknown/unspecified state. (Accessing a global variable is always unspecified state.)

[Andy] We've given "global" a variety of meanings. I think we should avoid that term unless specifically referring to global variables.

There are some function level effects that are not specific to state:

- allocs
- traps

The effects on a particular state are:

- read
- write
- capture
- release

These should all be interpreted as effects that "may" happen. e.g. maywrite, or mayretain.

[TODO] Within the optimizer we sometimes refer to "retain" as an effect. "capture" is really a more general term that encompasses any retain_value operation, so we'll likely standardize on that term. We don't have a more general term for "release", which refers to any release_value operation.

When referring to unspecified state, I will use the syntax `@_effects(no<effectname>)`. When referring to state reachable via an argument, `@no<effectname> arg`.

Naturally, we also need a syntax for associating effects with `self`. That could easily be done by adding a @self_effects attribute.

In order to optimize bridged types, we need to add a `nonbridged` predicate to the effects. The optimizer can then reason about a value's bridged status within some scope and deduce more optimistic effects at a call site. For now, we assume the predicate only applies to unspecified state and that the bridged object is always self. That way we can denote predicated effects as @nonbridged_effects.

In examples, @_effects(argonly) means that there are no effects on unspecified state.

CoW Optimization Requirements

Swift-level attributes proposal

A copy-on-write (COW) type is implemented in terms of a struct and a set of storage objects referenced by this struct. The set of storage objects can further provide storage for subobjects.:

```
class ArrayStorage<T> {
  func getElement(_ index: Int) -> T {} // Return a 'subobject'.
}

struct Array<T> {
  var storage: ArrayStorage // Storage object
}
```

In the following we will list a set of function attributes that can be used to describe properties of methods of such a data structure to facilitate optimization.

A COW type implements value semantics by delaying the copy of storage of the type until modification.

An instance of a struct is in a uniqued state if changes to the set of storage objects can only be observed by method calls on references to the instance of the struct (versus by method calls on other instances). Typically, one would implement this behavior by checking whether the references to the storage objects are uniquely referenced and copying the storage objects on modification if they are not. In the following we refer to the memory holding the instance of the struct and the set of storage objects as the self state. Non-self state below refers to the state of the rest of the program not including the self state.

`@make_unique`

> A method marked `@make_unique` changes the state of the instance of the COW type (`self`) to the uniqued state. It must do so without changing or depending on non-self state or changing the self-state (other than the change to a uniqued state). It must be an idempotent operation.:
>
> ```
> struct Array<T> {
>   var storage: ArrayStorage
>
>   @makeunique
>   mutating func makeUnique() {
>     if (isUniquelyReferenced(&storage))
>       return
>     storage = storage.copy()
>   }
> ```
>
> Note: In terms of low-level SIL attributes such a method will be marked::
>
> ```
> @_effects(argonly)
> @selfeffects(make_unique)
> func makeUnique() {}
> ```

`@preserve_unique`

A method marked `@preserve_unique` must guarantee to not change the uniqueness state of `self` from a unique state to a not unique state. An example of a violation of this guarantee would be to store `self` in a global variable. The method must not return a storage object or address there-of that could be used to change the uniqueness state of `self`. An example of a violation of this guarantee would be a method that returns a storage object.:

```
struct Array<T> {
  var storage: ArrayStorage

  @preserve_unique
  mutating func replaceSubrange<
    C : CollectionType where C.Iterator.Element == T
  >(
    _ subRange: Range<Int>, with newElements: C
  ) { ... }

  // We could also mark the following function as @preserve_unique
  // but we have an attribute for this function that better describes it
  // allowing for more optimization. (See @get_subobject)
  @preserve_unique
  func getElement(_ index: Int) -> T {
    return storage.elementAt(index)
  }
}
```

Note: In terms of low-level SIL attributes such a method will be marked::

```
@self_effects(preserve_unique, nocapture, norelease)
func replace<> {}
```

`@get_subobject`

A method marked `@get_subobject` must fulfill all of `@preserve_unique`'s guarantees. Furthermore, it must return a 'subobject' that is stored by the set of storage objects or a value stored in the CoW struct itself. It must be guaranteed that the 'subobject' returned is kept alive as long the current value of the 'self' object is alive. Neither the self state nor the non-self state is changed and the method must not depend on non-self state.:

```
struct Array<T> {
  var storage: ArrayStorage
  var size : Int

  @get_subobject
  func getElement(_ index: Int) -> T {
    return storage.elementAt(index)
  }

  @get_subobject
  func getSize() -> Int {
    return size
  }
```

Note: In terms of low-level SIL attributes such a method will be marked::

```
@_effects(argonly)
@selfeffects(preserve_unique, nowrite, nocapture, norelease,
             projects_subobject)
func getElement(_ index: Int) -> T {}
```

> **Note**
>
> For the standard library's data types `@get_subobject` guarantees are too strong. An array can use an NSArray as its storage (it is in a bridged state) in which case we can't make assumptions on effects on non-self state. For this purpose we introduce a variant of the attribute above whose statement about global effects are predicated on the array being in a non-bridged state.

`@get_subobject_non_bridged`

A method marked `@get_subobject` must fulfill all of `@preserve_unique`'s guarantees. Furthermore, it must return a 'subobject' that is stored by the set of storage objects or a value stored in the CoW struct itself. It must be guaranteed that the 'subobject' returned is kept alive as long the current value of the 'self' object is alive. The self state is not changed. The non-self state is not changed and the method must not depend on non-self state if the `self` is in a non-bridged state. In a bridged state the optimizer will assume that subsequent calls on the same 'self' object to return the same value and that consecutive calls are idempotent however it will not assume anything beyond this about effects on non-self state.:

```
struct Array<T> {
  var storage: BridgedArrayStorage
  var size : Int
```

```
  @get_subobject_non_bridged
  func getElement(_ index: Int) -> T {
    return storage.elementAt(index)
  }

  @get_subobject
  func getSize() -> Int {
    return size
  }
```

Note: In terms of low-level SIL attributes such a method will be marked::

```
@nonbridged_effects(argonly)
@selfeffects(preserve_unique, nowrite, nocapture, norelease,
             projects_subobject)
func getElement(_ index: Int) -> T {}
```

`@get_subobject_addr`

A method marked `@get_subobject_addr` must fulfill all of `@preserve_unique`'s guarantees. Furthermore, it must return the address of a 'subobject' that is stored by the set of storage objects. It is guaranteed that the 'subobject' at the address returned is kept alive as long the current value of the 'self' object is alive. Neither the self state nor the non-self state is changed and the method must not depend on non-self state.:

```
struct Array<T> {
  var storage: ArrayStorage

  @get_subobject_addr
  func getElementAddr(_ index: Int) -> UnsafeMutablePointer<T> {
    return storage.elementAddrAt(index)
  }
```

Note: In terms of low-level SIL attributes such a method will be marked::

```
@_effects(argonly)
@selfeffects(preserve_unique, nowrite, nocapture, norelease,
             projects_subobject_addr)
func getElementAddr(_ index: Int) -> T {}
```

`@initialize_subobject`

A method marked `@initialize_subobject` must fulfill all of `@preserve_unique`'s guarantees. The method must only store its arguments into *uninitialized* storage. The only effect to non-self state is the capture of the method's arguments.:

```
struct Array<T> {
  var storage: ArrayStorage

  @initialize_subobject
  func appendAssumingUniqueStorage(_ elt: T) {
    storage.append(elt)
  }
}
```

Note: In terms of low-level SIL attributes such a method will be marked::

```
@_effects(argonly)
@selfeffects(preserve_unique, nocapture, norelease)
func appendElementAssumingUnique(@norelease @nowrite elt: T) {}
```

> **Note**
>
> [arnold] We would like to express something like `@set_subobject`, too. However, we probably want to delay this until we have a polymorphic effects type system.

`@set_subobject`

A method marked `@set_subobject` must fulfill all of `@preserve_unique`'s guarantees. The method must only store its arguments into *initialized* storage. The only effect to non-self state is the capture of the method's arguments and the release of objects of the method arguments' types.:

```
struct Array<T> {
  var storage: ArrayStorage

  @set_subobject
  func setElement(_ elt: T, at index: Int) {
    storage.set(elt, index)
  }
```

```
}
```

Note

[arnold] As Andy points out, this would be best expressed using an effect type system.

Note: In terms of low-level SIL attributes such a method will be marked::

```
@_effects(argonly, T.release)
@selfeffects(preserve_unique, nocapture)
func setElement(@nowrite e: T, index: Int) {
}
```

Motivation

Why do we need `makeunique`, `preserveunique`?

The optimizer wants to hoist functions that make a COW type instance unique out of loops. In order to do that it has to prove that uniqueness is preserved by all operations in the loop.

Marking methods as `makeunique`/`preserveunique` allows the optimizer to reason about the behavior of the method calls.

Example::

```
struct Array<T> {
  var storage: ArrayStorage<T>

  @makeunique
  func makeUnique() {
    if (isUniquelyReferenced(&storage))
     return;
    storage = storage.copy()
  }

  @preserveunique
  func getElementAddr(_ index: Int) -> UnsafeMutablePointer<T> {
    return storage.elementAddrAt(index)
  }

  subscript(index: Int) -> UnsafeMutablePointer<T> {
    mutableAddressor {
      makeUnique()
      return getElementAddr(index)
    }
  }
}
```

When the optimizer optimizes a loop::

```
func memset(A: inout [Int], value: Int) {
  for i in 0 .. A.size {
    A[i] = value
    f()
  }
}
```

It will see the following calls because methods with attributes are not inlined.:

```
func memset(A: inout [Int], value: Int) {
  for i in 0 .. A.size {
    makeUnique(&A)
    addr = getElementAddr(i, &A)
    addr.pointee = value
    f()
  }
}
```

In order to hoist the 'makeUnique' call, the optimizer needs to be able to reason that neither 'getElementAddr', nor the store to the address returned can change the uniqueness state of 'A'. Furthermore, it knows because 'A' is marked inout that in a program without inout violations f cannot hold a reference to the object named by 'A' and therefore cannot modify it.

Why do we need `@get_subobject`, `@initialize_subobject`, and `@set_subobject`?

We want to be able to hoist `makeunique` calls when the array is not identified by a unique name.:

```
class AClass {
  var array: [Int]
}

func copy(_ a : AClass, b : AClass) {
```

```
    for i in min(a.size, b.size) {
       a.array.append(b.array[i])
    }
  }
```

In such a case we would like to reason that::

```
  = b.array[i]
```

cannot changed the uniqueness of the instance of array 'a.array' assuming 'a' !== 'b'. We can do so because 'getElement' is marked `@get_subobject` and so does not modify non-self state.

Further we would like to reason that::

```
  a.array.append
```

cannot change the uniqueness state of the instance of array 'a.array' across iterations. We can conclude so because `appendAssumingUnique`'s side-effects guarantee that no destructor can run - it's only side-effect is that `tmp` is captured and initializes storage in the array - these are the only side-effects according to `@initialize_subobject`.:

```
  for i in 0 .. b.size {
    // @get_subobject
    tmp = getElement(b.array, i)
    makeUnique(&a.array)
    // @initialize_subobject
    appendAssumingUnique(&a.array, tmp)
  }
```

We can construct a very similar example where we cannot hoist makeUnique. If we replace 'getElement' with a 'setElement'. 'setElement' will capture its argument and further releases an element of type T - these are the only side-effects according to `@set_subobject`:

```
  @set_subobject
  func setElement(_ e: T, index: Int) {
    storage->setElement(e, index)
  }
```

Depending on 'T's type a destructor can be invoked by the release of 'T'. The destructor can have arbitrary side-effects. Therefore, it is not valid to hoist the makeUnique in the code without proving that 'T's destructor cannot change the uniqueness state. This is trivial for trivial types but requires a more sophisticated analysis for class types (and in general cannot be disproved). In following example we can only hoist makeUnique if we can prove that elt's, and elt2's destructor can't change the uniqueness state of the arrays.:

```
  for i in 0 ..< min(a.size, b.size) {
    makeUnique(&b.array)
    setElement(&b.array, elt, i)
    makeUnique(&a.array)
    setElement(&a.array, elt2, i)
  }
```

In the following loop it is not safe to hoist the makeUnique(&a) call even for trivial types. 'appendAssumingUnique' captures its argument 'a' which forces a copy on 'a' on every iteration of the loop.:

```
  for i in 0 .. a.size {
    makeUnique(&a)
    setElement(&a, 0, i)
    makeUnique(&b)
    appendAssumingUnique(&b, a)
  }
```

To support this reasoning we need to know when a function captures its arguments and when a function might release an object and of which type.

`@get_subobject` and value-type behavior

Furthermore, methods marked with `@get_subobject` will allow us to remove redundant calls to read-only like methods on COW type instances assuming we can prove that the instance is not changed in between them.:

```
  func f(_ a: [Int]) {
   @get_subobject
   count(a)
   @get_subobject
   count(a)
  }
```

Examples of Optimization Using Effects Primitives

CoW optimization: [Let's copy over examples from Arnold's proposal]

[See the Copy-on-write proposal above]

String initialization: [TBD]

User-Specified Effects, Syntax and Defaults

Mostly TBD.

The optimizer can only take advantage of user-specified effects before they have been inlined. Consequently, the optimizer initially preserves calls to annotated @_effects() functions. After optimizing for effects these functions can be inlined, dropping the effects information.

Without special syntax, specifying a pure function would require:

```
@_effects(argonly)
func foo(@noread @nowrite arg)
```

A shorthand, such as @_effects(none) could easily be introduced. Typically, this shouldn't be needed because the purity of a function can probably be deduced from its argument types given that it has no effect on unspecified state. i.e. If the function does not affect unspecific state, and operates on "pure value types" (see below), the function is pure.

Specifying Effects for Generic Functions

Specifying literal function effects is not possible for functions with generic arguments:

```
struct MyContainer<T> {
  var t: T
  func setElt(_ elt: T) { t = elt }
}
```

With no knowledge of T.deinit() we must assume the worst case. SIL effects analysis following specialization can easily handle such a trivial example. But there are two situations to be concerned about:

1. Complicated CoW implementations defeat effects analysis. That is the whole point of Arnold's proposal for user-specified CoW effects.
2. Eventually we will want to publish effects on generic functions across resilience boundaries.

Solving this requires a system for polymorphic effects. Language support for polymorphic effects might look something like this:

```
@_effects(T.release)
func foo<T>(t: T) { ... }
```

This would mean that foo's unspecified effects are bounded by the unspecified effects of T's deinitializer. The reality of designing polymorphic effects will be much more complicated.

A different approach would be to statically constrain effects on generic types, protocol conformance, and closures. This wouldn't solve the general problem, but could be a very useful tool for static enforcement.

> **Note**
>
> Examples of function effects systems:
>
> [JoeG] For example, the effect type system model in Koka (https://koka.codeplex.com) can handle exceptions, side effects on state, and heap capture in polymorphic contexts in a pretty elegant way. It's my hope that "throws" can provide a seed toward a full effects system like theirs.
>
> http://www.eff-lang.org: A language with first-class effects.

Purity

Motivation for Pure Functions

An important feature of Swift structs is that they can be defined such that they have value semantics. The optimizer should then be able to reason about these types with knowledge of those value semantics. This in turn allows the optimizer to reason about function purity, which is a powerful property. In particular, calls to pure functions can be hoisted out of loops and combined with other calls taking the same arguments. Pure functions also have no detrimental effect on optimizing the surrounding code.

For example:

```
func bar<T>(t: T) {...}

func foo<T>(t: T, N: Int) {
  for _ in 1...N {
    bar(t)
    bar(t)
  }
}
```

With some knowledge of bar() and T can become:

```
func foo<T>(t: T, N: Int) {
  bar(t)
}
```

If our own implementation of value types, like Array, Set, and String where annotated as know "pure values" and if their common operations are known to comply with some low-level effects, then the optimizer could infer more general purity of operations on those types. The optimizer could then also reason about purity of operations on user defined types composed from Arrays, Sets, and Strings.

"Pure" Value Types

Conceptually, a pure value does not share state with another value. Any trivial struct is automatically pure. Other structs can be declared pure by the author. It then becomes the author's responsibility to guarantee value semantics. For instance, any stored reference into the heap must either be to immutable data or protected by CoW.

Since a pure value type can in practice share implementation state, we need an enforceable definition of such types. More formally:

- Copying or destroying a pure value cannot affect other program state.
- Reading memory referenced from a pure value does not depend on other program state. Writing memory referenced from a pure value cannot affect other program state.

The purity of functions that operate on these values, including their own methods, must be deduced independently.

From the optimizer perspective, there are two aspects of type purity that fall out of the definition:

1. Side Effects of Copies

   Incrementing a reference count is not considered a side effect at the level of value semantics. Destroying a pure value only destroys objects that are part of the value's storage. This could be enforced by prohibiting arbitrary code inside the storage deinitializer.

2. Aliasing

   Mutation of the pure value cannot affect program state apart from that value, AND writing program state outside the value cannot affect the pure value.

[Note] Reference counts are exposed through the isUniquelyReferenced API. Since copying a pure value can increase the reference of the storage, strictly speaking, a pure function can have user-visible side effects. We side step this issue by placing the burden on the user of the isUniquelyReferenced API. The compiler only guarantees that the API returns a non-unique reference count if there does happen to be an aliasing reference after optimization, which the user cannot control. The user must ensure that the program behaves identically in either case apart from its performance characteristics.

Pure Value Types and SIL optimizations

The benefit of having pure value types is that optimizations can treat such types as if they were Swift value types, like struct. Member functions of pure value types can be annotated with effects, like `readnone` for `getElement`, even if the underlying implementation of `getElement` reads memory from the type's storage.

The compiler can do more optimistic optimizations for pure value types without the need of sophisticated alias or escape analysis.

Consider this example.:

```
func add(_ arr: Array<Int>, i: Int) -> Int {
  let e1 = arr[i]
  unknownFunction()
  let e2 = arr[i]
}
```

This code is generated to something like:

```
func add(_ arr: Array<Int>, i: Int) -> Int {
  let e1 = getElement(i, arr)
  unknownFunction()
  let e2 = getElement(i, arr)
  return e1 + e2
}
```

Now if the compiler can assume that Array is a pure value type and `getElement` has a defined effect of `readnone`, it can CSE the two calls. This is because the arguments, including the `arr` itself, are the same for both calls.

Even if `unknownFunction` modifies an array which references the same storage as `arr`, CoW semantics will force `unknownFunction` to make a copy of the storage and the storage of `arr` will not be modified.

Pure value types can only considered pure on high-level SIL, before effects and semantics functions are inlined. For an example see below.

[TBD] Effects like `readnone` would have another impact on high-level SIL than on low-level SIL. We have to decide how we want to handle this.

Recognizing Value Types

Recognizing Value Types

A major difficulty in recognizing value types arises when those types are implemented in terms of unsafe code with arbitrary side effects. This is the crux of the difficulty in defining the CoW effects. Consequently, communicating purity to the compiler will require some function annotations and/or type constraints.

A CoW type consists of a top-level value type, most likely a struct, and a referenced storage, which may be shared between multiple instances of the CoW type.

[TBD] Is there any difference between a 'CoW type' and a 'pure value type'? E.g. can there be CoW types which are not pure value types or vice versa?

The important thing for a pure value type is that all functions which change the state are defined as mutating, even if they don't mutate the top-level struct but only the referenced storage.

> **Note**
>
> For CoW data types this is required anyway, because any state-changing function will have to unique the storage and thus be able to replace the storage reference in the top-level struct.

Let's assume we have a setElement function in Array.:

```
mutating func setElement(_ i: Int, e: Element) {
  storage[i] = e
}
```

Let's replace the call to `unknownFunction` with a set of the i'th element in our example. The mutating function forces the array to be placed onto the stack and reloaded after the mutating function. This lets the second `getElement` function get another array parameter which prevents CSE of the two `getElement` calls. Shown in this swift-SIL pseudo code:

```
func add(arr: Array<Int>, i: Int) -> Int {
  var arr = arr
  let e1 = getElement(i, arr)
  store arr to stack_array
  setElement(i, 0, &stack_array)
  let arr2 = load from stack_array
  let e2 = getElement(i, arr2)     // arr2 is another value than arr
  return e1 + e2
}
```

Another important requirement for pure value types is that all functions, which directly access the storage, are not inlined during high-level SIL. Optimizations like code motion could move a store to the storage over a `readnone getElement`.:

```
func add(arr: Array<Int>, i: Int) -> Int {
  var arr = arr
  let e1 = getElement(i, arr)
  store arr to stack_array
  stack_array.storage[i] = 0          // (1)
  let arr2 = load from stack_array    // (2)
  let e2 = getElement(i, arr2)        // (3)
  return e1 + e2
}
```

Store (1) and load (2) do not alias and (3) is defined as `readnone`. So (1) could be moved over (3).

Currently inlining is prevented in high-level SIL for all functions which have a semantics or effect attribute. Therefore we could say that the implementor of a pure value type has to define effects on all member functions which eventually can access or modify the storage.

To help the user to fulfill this contract, the compiler can check if some effects annotations are missing. For this, the storage properties of a pure value type should be annotated. The compiler can check if all call graph paths from the type's member functions to storage accessing functions contain at least one function with defined effects. Example:

```
struct Array {

  @cow_storage var storage

  @effect(...)
  func getElement() { return storage.get() }

  @effect(...)
  func checkSubscript() { ... }

  subscript { get {          // OK
    checkSubscript()
    return getElement()
  } }

  func getSize() {
```

```
        return storage.size()  // Error!
    }
  }
```

[TBD] What if a storage property is public. What if a non member function accesses the storage.

As discussed above, CoW types will often be generic, making the effects of an operation on the CoW type dependent on the effects of destroying an object of the element type.

[erik] This is not the case if CoW types are always passed as guaranteed to the effects functions.

Inferring Function Purity

The optimizer can infer function purity by knowing that (1) the function does not access unspecified state, (2) all arguments are pure values, and (3) no calls are made into non-pure code.

1. The effects system described above already tells the optimizer via analysis or annotation that the function does not access unspecified state.
2. Copying or destroying a pure value by definition has no impact on other program state. The optimizer may either deduce this from the type definition, or it may rely on a type constraint.
3. Naturally, any calls within the function body must be transitively pure. There is no need to check calls to the storage deinitializer, which should already be guaranteed pure by virtue of (2).

Mutability of a pure value should not affect the purity of functions that operate on the value. An inout argument is semantically nothing more than a copy of the value.

[Note] Pure functions do not depend on or imply anything about the reference counting effects: capture and release. Optimizations that depend on reference count stability, like uniqueness hoisting, cannot treat pure functions as side-effect free.

> **Note**
>
> [Andy] It may be possible to make some assumptions about immutability of `let` variables, which could lead to similar optimization.

TODO: Need more clarity and examples

Closures

Mostly TBD.

The optimizer does not currently have a way of statically determining or enforcing effects of a function that takes a closure. We could introduce attributes that statically enforce constraints. For example, and @pure closure would only be permitted to close over pure values.

> **Note**
>
> [Andy] That is a fairly strict requirement, but not one that I know how to overcome.

Thread Safety

The Swift concurrency proposal refers to a `Copyable` type. A type must be Copyable in order to pass it across threads via a `gateway`. The definition of a Copyable type is equivalent to a "pure value". However, it was also proposed that the programmer be able to annotate arbitrary data types as Copyable even if they contain shared state as long as it is protected via a mutex. However, such data types cannot be considered pure by the optimizer. I instead propose that a separate constraint, Synchronized, be attributed to shareable types that are not pure. An object could be passed through a gateway either if it is a PureValue or is Synchronized.

Annotations for thread safety run into the same problems with generics and closures.

API and Resilience

Any type constraints, function effects, or closure attributes that we introduce on public functions become part of the API.

Naturally, there are resilience implications to user-specified effects. Moving to a weaker set of declared effects is not resilient.

Generally, a default-safe policy provides a much better user model from some effects. For example, we could decide that functions cannot affect unspecified state by default. If the user accesses globals, they then need to annotate their function. However, default safety dictates that any necessary annotations should be introduced before declaring API stability.