orphan:

High-Level Optimizations in SIL

Contents

Abstract

This document describes the high-level abstraction of built-in Swift data structures in SIL that is used by the optimizer. You need to read this document if you wish to understand the early stages of the Swift optimizer or if you are working on one of the containers in the standard library.

Why do we need high-level optimizations?

Swift containers are implemented in the Swift standard library in Swift code. Traditional compiler optimizations can remove some of the redundancy that is found in high-level code, but not all of it. Without knowledge of the Swift language the optimizer can't perform high-level optimizations on the built-in containers. For example:

```
Dict["a"] = 1
Dict["a"] = 2
```

Any Swift developer could identify the redundancy in the code sample above. Storing two values into the same key in the dictionary is inefficient. However, optimizing compilers are unaware of the special semantics that the Swift dictionary has and can't perform this optimization. Traditional compilers would start optimizing this code by inlining the subscript function call and try to analyze the sequence of load/store instructions. This approach is not very effective because the compiler has to be very conservative when optimizing general code with pointers.

On the other hand, compilers for high-level languages usually have special bytecode instructions that allow them to perform high-level optimizations. However, unlike high-level languages such as JavaScript or Python, Swift containers are implemented in Swift itself. Moreover, it is beneficial to be able to inline code from the container into the user program and optimize them together, especially for code that uses Generics.

In order to perform both high-level optimizations, that are common in high-level languages, and low-level optimizations we annotate parts of the standard library and describe the semantics of a domain-specific high-level operations on data types in the Swift standard library.

Annotation of code in the standard library

We use the `@_semantics` attribute to annotate code in the standard library. These annotations can be used by the high-level SIL optimizer to perform domain-specific optimizations. The same function may have multiple `@_semantics` attributes.

This is an example of the `@_semantics` attribute:

```
@public @_semantics("array.count")
func getCount() -> Int {
  return _buffer.count
 }
```

In this example we annotate a member of the Swift array struct with the tag `array.count`. This tag informs the optimizer that this method reads the size of the array.

The `@_semantics` attribute allows us to define "builtin" SIL-level operations implemented in Swift code. In SIL code they are encoded as apply instructions, but the optimizer can operate on them as atomic instructions. The semantic annotations don't necessarily need to be on public APIs. For example, the Array subscript operator may invoke two operations in the semantic model. One for checking the bounds and another one for accessing the elements. With this abstraction the optimizer can remove the `checkSubscript` instruction and keep the getElement instruction:

```
@public subscript(index: Int) -> Element {
   get {
    checkSubscript(index)
    return getElement(index)
   }

@_semantics("array.check_subscript") func checkSubscript(_ index: Int) {
  ...
}

@_semantics("array.get_element") func getElement(_ index: Int) -> Element {
  return _buffer[index]
}
```

Swift optimizations

The Swift optimizer can access the information that is provided by the `@_semantics` attribute to perform high-level optimizations. In the early stages of the optimization pipeline the optimizer does not inline functions with special semantics in order to allow the early high-level optimization passes to operate on them. In the later stages of the optimization pipeline the optimizer inlines functions with special semantics to allow low-level optimizations.

Annotated data structures in the standard library

This section describes the semantic tags that are assigned to data-structures in the standard library and the axioms that the optimizer uses.

Cloning code from the standard library

The Swift compiler can copy code from the standard library into the application for functions marked @inlinable. This allows the optimizer to inline calls from the stdlib and improve the performance of code that uses common operators such as '+=' or basic containers such as Array. However, importing code from the standard library can increase the binary size.

To prevent copying of functions from the standard library into the user program, make sure the function in question is not marked @inlinable.

Array

The following semantic tags describe Array operations. The operations are first described in terms of the Array "state". Relations between the operations are formally defined below. 'Array' refers to the standard library Array<Element>, ContiguousArray<Element>, and ArraySlice<Element> data-structures.

We consider the array state to consist of a set of disjoint elements and a storage descriptor that encapsulates non-element data such as the element count and capacity. Operations that semantically write state are always *control dependent*. A control dependent operation is one that may only be executed on the control flow paths in which the operation originally appeared, ignoring potential program exits. Generally, operations that only read state are not control dependent. One exception is `check_subscript` which is readonly but control dependent because it may trap. Some operations are *guarded* by others. A guarded operation can never be executed before its guard.

array.init

> Initialize an array with new storage. This currently applies to any initializer that does not get its storage from an argument. This semantically writes to every array element and the array's storage descriptor. `init` also implies the guarding semantics of `make_mutable`. It is not itself guarded by `make_mutable` and may act as a guard to other potentially mutating operations, such as `get_element_address`.

array.uninitialized(count: Builtin.Word) -> (Array<Element>, Builtin.RawPointer)

> Creates an array with the specified number of elements. It initializes the storage descriptor but not the array elements. The returned tuple contains the new array and a raw pointer to the element storage. The caller is responsible for writing the elements to the element storage.

array.props.isCocoa/needsElementTypeCheck -> Bool

> Reads storage descriptors properties (isCocoa, needsElementTypeCheck). This is not control dependent or guarded. The optimizer has semantic knowledge of the state transfer those properties cannot make: An array that is not `isCocoa` cannot transfer to `isCocoa`. An array that is not `needsElementTypeCheck` cannot transfer to `needsElementTypeCheck`.

array.get_element(index: Int) -> Element

> Read an element from the array at the specified index. No other elements are read. The storage descriptor is not read. No state is written. This operation is not control dependent, but may be guarded by `check_subscript`. Any `check_subscript` may act as a guard, regardless of the index being checked [1].

array.get_element_address(index: Int) -> UnsafeMutablePointer<Element>

Get the address of an element of the array. No state is written. The storage descriptor is not read. The resulting pointer may be used to access elements in the array. This operation is not control dependent, but may be guarded by `check_subscript`. Any `check_subscript`, `make_mutable` or `mutate_unknown` may act as a guard.

array.check_subscript(index: Int)

> Read the array count from the storage descriptor. Execute a `trap` if `index < array.startIndex || index >= array.endIndex`. No elements are read. No state is written. Despite being read only, this operation is control dependent.

array.get_count() -> Int

> Read the array count (`array.endIndex - array.startIndex`) from the storage descriptor. No elements are read. No state is written. This is neither guarded nor control dependent.

array.get_capacity() -> Int

> Read the array capacity from the storage descriptor. The semantics are identical to `get_count` except for the meaning of the return value.

array.append_element(newElement: Element)

> Appends a single element to the array. No elements are read. The operation is itself guarded by `make_mutable`. In contrast to other semantics operations, this operation is allowed to be inlined in the early stages of the compiler.

array.append_contentsOf(contentsOf newElements: S)

> Appends all elements from S, which is a Sequence. No elements are read. The operation is itself guarded by `make_mutable`.

array.make_mutable()

> This operation guards mutating operations that don't already imply `make_mutable` semantics. (Currently, the only guarded operation is `get_element_address`.) `make_mutable` may create a copy of the array storage; however, semantically it neither reads nor writes the array state. It does not write state simply because the copy's state is identical to the original. It does not read state because no other Array operations can undo mutability--only code that retains a reference to the Array can do that. `make_mutable` does effectively need to be guarded by any SIL operation that may retain the array. Because `make_mutable` semantically does not read the array state, is idempotent, and has no control dependence, it can be executed safely on any array at any point. i.e. the optimizer can freely insert calls to make_mutable.

array.mutate_unknown

> This operation may mutate the array in any way, so it semantically writes to the entire array state and is naturally control dependent. `mutate_unknown` also implies the guarding semantics of `make_mutable`. It is not itself guarded by `make_mutable` and may act as a guard to other mutating operations, such as `get_element_address`. Combining semantics allows the flexibility in how the array copy is implemented in conjunction with implementing mutating functionality. This may be more efficient than cleanly isolating the copy and mutation code.

To complete the semantics understood by the optimizer, we define these relations:

interferes-with

> Given idempotent `OpA`, the sequence "`OpA, OpB, OpA`" is semantically equivalent to the sequence "`OpA, OpB`" *iff* `OpB` does not interfere with `OpA`.
>
> All array operations marked with semantics are idempotent as long as they call the same function with the same argument values, with the exception of `mutate_unknown`.

guards

> If `OpA` guards `OpB`, then the sequence of operations `OpA, OpB` must be preserved on any control flow path on which the sequence originally appears.

An operation can only interfere-with or guard another if they may operate on the same Array. `get_element_address` is abbreviated with `get_elt_addr` in the table below.

semantic op	**relation**	**semantic ops**
make_mutable	guards	get_element_address
check_subscript	guards	get_element, get_element_address
make_mutable	interferes-with	props.isCocoa/needsElementTypeCheck

semantic op	relation	semantic ops
get_elt_addr	interferes-with	get_element, get_element_address, props.isCocoa/needsElementTypeCheck
mutate_unknown	interferes-with	get_element, check_subscript, get_count, get_capacity, get_element_address, props.isCocoa/needsElementTypeCheck

[1] Any check_subscript(N) may act as a guard for `get_element(i)/get_element_address(i)` as long as it can be shown that `N >= i`.

In addition to preserving these semantics, the optimizer must conservatively handle any unknown access to the array object. For example, if a SIL operation takes the address to any member of the Array, any subsequent operations that may have visibility of that address are considered to interfere with any array operations with explicit semantics.

String

string.concat(lhs: String, rhs: String) -> String

Performs concatenation of two strings. Operands are not mutated. This operation can be optimized away in case of both operands being string literals. In this case, it can be replaced by a string literal representing a concatenation of both operands.

string.makeUTF8(start: RawPointer, utf8CodeUnitCount: Word, isASCII: Int1) -> String

Converts a built-in UTF8-encoded string literal into a string.

string.makeUTF16(start: RawPointer, utf16CodeUnitCount: Word) -> String

Converts a built-in UTF16-encoded string literal into a string.

Dictionary

TBD.

@_effects attribute

The @_effects attribute describes how a function affects "the state of the world". More practically how the optimizer can modify the program based on information that is provided by the attribute.

Usage:

@_effects(readonly) func foo() { .. }

The @_effects attribute supports the following tags:

readnone

function has no side effects and no dependencies on the state of the program. It always returns an identical result given identical inputs. Calls to readnone functions can be eliminated, reordered, and folded arbitrarily.

readonly

function has no side effects, but is dependent on the global state of the program. Calls to readonly functions can be eliminated, but cannot be reordered or folded in a way that would move calls to the readnone function across side effects.

releasenone

function has side effects, it can read or write global state, or state reachable from its arguments. It can however be assumed that no externally visible release has happened (i.e it is allowed for a `releasenone` function to allocate and destruct an object in its implementation as long as this is does not cause an release of an object visible outside of the implementation). Here are some examples:

```
class SomeObject {
  final var x: Int = 3
}

var global = SomeObject()

class SomeOtherObject {
  var x: Int = 2
  deinit {
    global = SomeObject()
  }
}
```

```
@_effects(releasenone)
func validReleaseNoneFunction(x: Int) -> Int {
  global.x = 5
  return x + 2
}

@_effects(releasenone)
func validReleaseNoneFunction(x: Int) -> Int {
  var notExternallyVisibleObject = SomeObject()
  return x +  notExternallyVisibleObject.x
}

func notAReleaseNoneFunction(x: Int, y: SomeObject) -> Int {
  return x + y.x
}

func notAReleaseNoneFunction(x: Int) -> Int {
  var releaseExternallyVisible = SomeOtherObject()
  return x + releaseExternallyVisible.x
}
```

readwrite

function has side effects and the optimizer can't assume anything.

Optimize semantics attribute

The optimize attribute adds function-specific directives to the optimizer.

The optimize attribute supports the following tags:

sil.specialize.generic.never

The sil optimizer should never create generic specializations of this function.

optimize.sil.specialize.generic.partial.never

The sil optimizer should never create generic partial specializations of this function.

Availability checks

The availability attribute is used for functions which implement the `if #available` guards.

The availability attribute supports the following tags:

availability.osversion(major: Builtin.Word, minor: Builtin.Word, patch: Builtin.Word) -> Builtin.Int1

Returns true if the OS version matches the parameters.