Swift Intermediate Language (SIL)

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)SIL.rst, line 1)

Unknown directive type "highlight".

```
.. highlight:: none
```

Contents

- Debug Information
  - debug_value
- Accessing Memory
  - load
  - store
  - load_borrow
  - store_borrow
  - begin_borrow
  - end_borrow
  - end_lifetime
  - assign
  - assign_by_wrapper
  - mark_uninitialized
  - mark_function_escape
  - mark_uninitialized_behavior
  - copy_addr
  - destroy_addr
  - index_addr
  - tail_addr
  - index_raw_pointer
  - bind_memory
  - rebind_memory
  - begin_access
  - end_access
  - begin_unpaired_access
  - end_unpaired_access
- Reference Counting
  - strong_retain
  - strong_release
  - set_deallocating
  - strong_copy_unowned_value
  - strong_retain_unowned
  - unowned_retain
  - unowned_release
  - load_weak
  - store_weak
  - load_unowned
  - store_unowned
  - fix_lifetime
  - mark_dependence
  - is_unique
  - begin_cow_mutation
  - end_cow_mutation
  - is_escaping_closure
  - copy_block
  - copy_block_without_escaping
  - builtin "unsafeGuaranteed"
  - builtin "unsafeGuaranteedEnd"
- Literals
  - function_ref
  - dynamic_function_ref
  - prev_dynamic_function_ref
  - global_addr
  - global_value
  - integer_literal
  - float_literal
  - string_literal
  - base_addr_for_offset
- Dynamic Dispatch
  - class_method
  - objc_method
  - super_method
  - objc_super_method
  - witness_method
- Function Application
  - apply
  - begin_apply
  - abort_apply
  - end_apply
  - partial_apply
  - builtin
- Metatypes
  - metatype
  - value_metatype
  - existential_metatype
  - objc_protocol
- Aggregate Types
  - retain_value
  - retain_value_addr
  - unmanaged_retain_value
  - strong_copy_unmanaged_value
  - copy_value
  - explicit_copy_value
  - move_value
  - release_value
  - release_value_addr
  - unmanaged_release_value
  - destroy_value
  - autorelease_value
  - tuple
  - tuple_extract
  - tuple_element_addr
  - destructure_tuple
  - struct
  - struct_extract
  - struct_element_addr
  - destructure_struct
  - object
  - ref_element_addr
  - ref_tail_addr
- Enums
  - enum
  - unchecked_enum_data
  - init_enum_data_addr

Abstract

SIL is an SSA-form IR with high-level semantic information designed to implement the Swift programming language. SIL accommodates the following use cases:

- A set of guaranteed high-level optimizations that provide a predictable baseline for runtime and diagnostic behavior.
- Diagnostic dataflow analysis passes that enforce Swift language requirements, such as definitive initialization of variables and constructors, code reachability, switch coverage.
- High-level optimization passes, including retain/release optimization, dynamic method devirtualization, closure inlining, promoting heap allocations to stack allocations, promoting stack allocations to SSA registers, scalar replacement of aggregates (splitting aggregate allocations into multiple smaller allocations), and generic function instantiation.
- A stable distribution format that can be used to distribute "fragile" inlineable or generic code with Swift library modules, to be optimized into client binaries.

In contrast to LLVM IR, SIL is a generally target-independent format representation that can be used for code distribution, but it can also express target-specific concepts as well as LLVM can.

For more information on developing the implementation of SIL and SIL passes, see:

- SILProgrammersManual.md.
- SILFunctionConventions.md.
- SILMemoryAccess.md.

SIL in the Swift Compiler

At a high level, the Swift compiler follows a strict pipeline architecture:

- The *Parse* module constructs an AST from Swift source code.
- The *Sema* module type-checks the AST and annotates it with type information.
- The *SILGen* module generates *raw SIL* from an AST.
- A series of *Guaranteed Optimization Passes* and *Diagnostic Passes* are run over the raw SIL both to perform optimizations and to emit language-specific diagnostics. These are always run, even at -Onone, and produce *canonical SIL*.
- General SIL *Optimization Passes* optionally run over the canonical SIL to improve performance of the resulting executable. These are enabled and controlled by the optimization level and are not run at -Onone.
- *IRGen* lowers canonical SIL to LLVM IR.
- The LLVM backend (optionally) applies LLVM optimizations, runs the LLVM code generator and emits binary code.

The stages pertaining to SIL processing in particular are as follows:

SILGen

SILGen produces *raw SIL* by walking a type-checked Swift AST. The form of SIL emitted by SILGen has the following properties:

- Variables are represented by loading and storing mutable memory locations instead of being in strict SSA form. This is similar to the initial `alloca`-heavy LLVM IR emitted by frontends such as Clang. However, Swift represents variables as reference-counted "boxes" in the most general case, which can be retained, released, and captured into closures.
- Dataflow requirements, such as definitive assignment, function returns, switch coverage (TBD), etc. have not yet been enforced.
- `transparent` function optimization has not yet been honored.

These properties are addressed by subsequent guaranteed optimization and diagnostic passes which are always run against the raw SIL.

Guaranteed Optimization and Diagnostic Passes

After SILGen, a deterministic sequence of optimization passes is run over the raw SIL. We do not want the diagnostics produced by the compiler to change as the compiler evolves, so these passes are intended to be simple and predictable.

- **Mandatory inlining** inlines calls to "transparent" functions.
- **Memory promotion** is implemented as two optimization phases, the first of which performs capture analysis to promote `alloc_box` instructions to `alloc_stack`, and the second of which promotes non-address-exposed `alloc_stack` instructions to SSA registers.
- **Constant propagation** folds constant expressions and propagates the constant values. If an arithmetic overflow occurs during the constant expression computation, a diagnostic is issued.
- **Return analysis** verifies that each function returns a value on every code path and doesn't "fall off the end" of its definition, which is an error. It also issues an error when a `noreturn` function returns.
- **Critical edge splitting** splits all critical edges from terminators that don't support arbitrary basic block arguments (all non cond_branch terminators).

If all diagnostic passes succeed, the final result is the *canonical SIL* for the program.

TODO:

- Generic specialization
- Basic ARC optimization for acceptable performance at -Onone.

General Optimization Passes

SIL captures language-specific type information, making it possible to perform high-level optimizations that are difficult to perform on LLVM IR.

- **Generic Specialization** analyzes specialized calls to generic functions and generates new specialized version of the functions. Then it rewrites all specialized usages of the generic to a direct call of the appropriate specialized function.
- **Witness and VTable Devirtualization** for a given type looks up the associated method from a class's vtable or a type witness table and replaces the indirect virtual call with a call to the mapped function.
- **Performance Inlining**
- **Reference Counting Optimizations**
- **Memory Promotion/Optimizations**
- **High-level domain specific optimizations** The Swift compiler implements high-level optimizations on basic Swift containers such as Array or String. Domain specific optimizations require a defined interface between the standard library and the optimizer. More details can be found here: HighLevelSILOptimizations

Syntax

SIL is reliant on Swift's type system and declarations, so SIL syntax is an extension of Swift's. A `.sil` file is a Swift source file with added SIL definitions. The Swift source is parsed only for its declarations; Swift `func` bodies (except for nested declarations) and top-level code are ignored by the SIL parser. In a `.sil` file, there are no implicit imports; the `swift` and/or `Builtin` standard modules must be imported explicitly if used.

Here is an example of a `.sil` file:

```
sil_stage canonical

import Swift

// Define types used by the SIL function.

struct Point {
  var x : Double
  var y : Double
}

class Button {
  func onClick()
  func onMouseDown()
  func onMouseUp()
}

// Declare a Swift function. The body is ignored by SIL.
func taxicabNorm(_ a:Point) -> Double {
  return a.x + a.y
}

// Define a SIL function.
// The name @_T5norms11taxicabNormfT1aV5norms5Point_Sd is the mangled name
// of the taxicabNorm Swift function.
sil @_T5norms11taxicabNormfT1aV5norms5Point_Sd : $(Point) -> Double {
bb0(%0 : $Point):
  // func Swift.+(Double, Double) -> Double
  %1 = function_ref @_TsoilpfTSdSd_Sd
  %2 = struct_extract %0 : $Point, #Point.x
  %3 = struct_extract %0 : $Point, #Point.y
  %4 = apply %1(%2, %3) : $(Double, Double) -> Double
  return %4 : Double
}

// Define a SIL vtable. This matches dynamically-dispatched method
// identifiers to their implementations for a known static class type.
sil_vtable Button {
  #Button.onClick: @_TC5norms6Button7onClickfS0_FT_T_
  #Button.onMouseDown: @_TC5norms6Button11onMouseDownfS0_FT_T_
  #Button.onMouseUp: @_TC5norms6Button9onMouseUpfS0_FT_T_
}
```

SIL Stage

```
decl ::= sil-stage-decl
sil-stage-decl ::= 'sil_stage' sil-stage

sil-stage ::= 'raw'
sil-stage ::= 'canonical'
```

There are different invariants on SIL depending on what stage of processing has been applied to it.

- **Raw SIL** is the form produced by SILGen that has not been run through guaranteed optimizations or diagnostic passes. Raw SIL may not have a fully-constructed SSA graph. It may contain dataflow errors. Some instructions may be represented in non-canonical forms, such as `assign` and `destroy_addr` for non-address-only values. Raw SIL should not be used for native code generation or distribution.
- **Canonical SIL** is SIL as it exists after guaranteed optimizations and diagnostics. Dataflow errors must be eliminated, and certain instructions must be canonicalized to simpler forms. Performance optimization and native code generation are derived from this form, and a module can be distributed containing SIL in this (or later) forms.

SIL files declare the processing stage of the included SIL with one of the declarations `sil_stage raw` or `sil_stage canonical` at top level. Only one such declaration may appear in a file.

SIL Types

```
sil-type ::= '$' '*'? generic-parameter-list? type
```

SIL types are introduced with the `$` sigil. SIL's type system is closely related to Swift's, and so the type after the `$` is parsed largely according to Swift's type grammar.

Type Lowering

A *formal type* is the type of a value in Swift, such as an expression result. Swift's formal type system intentionally abstracts over a large number of representational issues like ownership transfer conventions and directness of arguments. However, SIL aims to represent most such implementation details, and so these differences deserve to be reflected in the SIL type system. *Type lowering* is the process of turning a formal type into its *lowered type*.

It is important to be aware that the lowered type of a declaration need not be the lowered type of the formal type of that declaration. For example, the lowered type of a declaration reference:

- will usually be thin,
- may have a non-Swift calling convention,
- may use bridged types in its interface, and
- may use ownership conventions that differ from Swift's default conventions.

Abstraction Difference

Generic functions working with values of unconstrained type must generally work with them indirectly, e.g. by allocating sufficient memory for them and then passing around pointers to that memory. Consider a generic function like this:

```
func generateArray<T>(n : Int, generator : () -> T) -> [T]
```

The function `generator` will be expected to store its result indirectly into an address passed in an implicit parameter. There's really just no reasonable alternative when working with a value of arbitrary type:

- We don't want to generate a different copy of `generateArray` for every type `T`.
- We don't want to give every type in the language a common representation.
- We don't want to dynamically construct a call to `generator` depending on the type `T`.

But we also don't want the existence of the generic system to force inefficiencies on non-generic code. For example, we'd like a function of type `() -> Int` to be able to return its result directly; and yet, `() -> Int` is a valid substitution of `() -> T`, and a caller of `generateArray<Int>` should be able to pass an arbitrary `() -> Int` in as the generator.

Therefore, the representation of a formal type in a generic context may differ from the representation of a substitution of that formal type. We call such differences *abstraction differences*.

SIL's type system is designed to make abstraction differences always result in differences between SIL types. The goal is that a properly- abstracted value should be correctly usable at any level of substitution.

In order to achieve this, the formal type of a generic entity should always be lowered using the abstraction pattern of its unsubstituted formal type. For example, consider the following generic type:

```
struct Generator<T> {
  var fn : () -> T
}
var intGen : Generator<Int>
```

`intGen.fn` has the substituted formal type `() -> Int`, which would normally lower to the type `@callee_owned () -> Int`, i.e. returning its result directly. But if that type is properly lowered with the pattern of its unsubstituted type `() -> T`, it becomes `@callee_owned () -> @out Int`.

When a type is lowered using the abstraction pattern of an unrestricted type, it is lowered as if the pattern were replaced with a type sharing the same structure but replacing all materializable types with fresh type variables.

For example, if `g` has type `Generator<(Int, Int) -> Float>`, `g.fn` is lowered using the pattern `() -> T`, which eventually causes `(Int, Int) -> Float` to be lowered using the pattern `T`, which is the same as lowering it with the pattern `U -> V`; the result is that `g.fn` has the following lowered type:

```
@callee_owned () -> @owned @callee_owned (@in (Int, Int)) -> @out Float.
```

As another example, suppose that `h` has type `Generator<(Int, inout Int) -> Float>`. Neither `(Int, inout Int)` nor `inout Int` are potential results of substitution because they aren't materializable, so `h.fn` has the following lowered type:

```
@callee_owned () -> @owned @callee_owned (@in Int, @inout Int) -> @out Float
```

This system has the property that abstraction patterns are preserved through repeated substitutions. That is, you can consider a lowered type to encode an abstraction pattern; lowering `T` by `R` is equivalent to lowering `T` by (`S` lowered by `R`).

SILGen has procedures for converting values between abstraction patterns.

At present, only function and tuple types are changed by abstraction differences.

Legal SIL Types

The type of a value in SIL is either:

- an *object type* `$T`, where `T` is a legal loadable type, or
- an *address type* `$*T`, where `T` is a legal SIL type (loadable or address-only).

A type `T` is a *legal SIL type* if:

- it is a function type which satisfies the constraints (below) on function types in SIL,
- it is a metatype type which describes its representation,
- it is a tuple type whose element types are legal SIL types,
- it is `Optional<U>`, where `U` is a legal SIL type,
- it is a legal Swift type that is not a function, tuple, optional, metatype, or l-value type, or
- it is a `@box` containing a legal SIL type.

Note that types in other recursive positions in the type grammar are still formal types. For example, the instance type of a metatype or the type arguments of a generic type are still formal Swift types, not lowered SIL types.

Address Types

The *address of T* `$*T` is a pointer to memory containing a value of any reference or value type `$T`. This can be an internal pointer into a data structure. Addresses of loadable types can be loaded and stored to access values of those types.

Addresses of address-only types (see below) can only be used with instructions that manipulate their operands indirectly by address, such as `copy_addr` or `destroy_addr`, or as arguments to functions. It is illegal to have a value of type `$T` if `T` is address-only.

Addresses are not reference-counted pointers like class values are. They cannot be retained or released.

Address types are not *first-class*: they cannot appear in recursive positions in type expressions. For example, the type `$**T` is not a legal type.

The address of an address cannot be directly taken. `$**T` is not a representable type. Values of address type thus cannot be allocated, loaded, or stored (though addresses can of course be loaded from and stored to).

Addresses can be passed as arguments to functions if the corresponding parameter is indirect. They cannot be returned.

Box Types

Captured local variables and the payloads of `indirect` value types are stored on the heap. The type `@box T` is a reference-counted type that references a box containing a mutable value of type `T`. Boxes always use Swift-native reference counting, so they can be queried for uniqueness and cast to the `Builtin.NativeObject` type.

Metatype Types

A concrete or existential metatype in SIL must describe its representation. This can be:

- `@thin`, meaning that it requires no storage and thus necessarily represents an exact type (only allowed for concrete metatypes);
- `@thick`, meaning that it stores a reference to a type or (if a concrete class) a subclass of that type; or
- `@objc`, meaning that it stores a reference to a class type (or a subclass thereof) using an Objective-C class object representation rather than the native Swift type-object representation.

Function Types

Function types in SIL are different from function types in Swift in a number of ways:

- A SIL function type may be generic. For example, accessing a generic function with `function_ref` will give a value of generic function type.
- A SIL function type may be declared `@noescape`. This is required for any function type passed to a parameter not declared with `@escaping` declaration modifier. `@noescape` function types may be either `@convention(thin)` or `@callee_guaranteed`. They have an unowned context--the context's lifetime must be independently guaranteed.
- A SIL function type declares its conventional treatment of its context value:
  - If it is `@convention(thin)`, the function requires no context value. Such types may also be declared `@noescape`, which trivially has no effect passing the context value.
  - If it is `@callee_guaranteed`, the context value is treated as a direct parameter. This implies `@convention(thick)`. If the function type is also `@noescape`, then the context value is unowned, otherwise it is guaranteed.
  - If it is `@callee_owned`, the context value is treated as an owned direct parameter. This implies `@convention(thick)` and is mutually exclusive with `@noescape`.
  - If it is `@convention(block)`, the context value is treated as an unowned direct parameter.
  - Other function type conventions are described in `Properties of Types` and `Calling Convention`.
- A SIL function type declares the conventions for its parameters. The parameters are written as an unlabeled tuple; the elements of that tuple must be legal SIL types, optionally decorated with one of the following convention attributes.

  The value of an indirect parameter has type `*T`; the value of a direct parameter has type `T`.

  - An `@in` parameter is indirect. The address must be of an initialized object; the function is responsible for destroying the value held there.
  - An `@inout` parameter is indirect. The address must be of an initialized object. The memory must remain initialized for the duration of the call until the function returns. The function may mutate the pointee, and furthermore may weakly assume that there are no aliasing reads from or writes to the argument, though must preserve a valid value at the argument so that well-ordered aliasing violations do not compromise memory safety. This allows for optimizations such as local load and store propagation, introduction or elimination of temporary copies, and promotion of the `@inout` parameter to an `@owned` direct parameter and result pair, but does not admit "take" optimization out of the parameter or other optimization that would leave memory in an uninitialized state.
  - An `@inout_aliasable` parameter is indirect. The address must be of an initialized object. The memory must remain initialized for the duration of the call until the function returns. The function may mutate the pointee, and must assume that other aliases may mutate it as well. These aliases however can be assumed to be well-typed and well-ordered; ill-typed accesses and data races to the parameter are still undefined.
  - An `@owned` parameter is an owned direct parameter.
  - A `@guaranteed` parameter is a guaranteed direct parameter.
  - An `@in_guaranteed` parameter is indirect. The address must be of an initialized object; both the caller and callee promise not to mutate the pointee, allowing the callee to read it.
  - An `@in_constant` parameter is indirect. The address must be of an initialized object; the function will treat the value held there as read-only.
  - Otherwise, the parameter is an unowned direct parameter.
- A SIL function type declares the conventions for its results. The results are written as an unlabeled tuple; the elements of that tuple must be legal SIL types, optionally decorated with one of the following convention attributes. Indirect and direct results may be interleaved.

  Indirect results correspond to implicit arguments of type `*T` in function entry blocks and in the arguments to `apply` and `try_apply` instructions. These arguments appear in the order in which they appear in the result list, always before any parameters.

  Direct results correspond to direct return values of type `T`. A SIL function type has a `return type` derived from its direct results in the following way: when there is a single direct result, the return type is the type of that result; otherwise, it is the tuple type of the types of all the direct results, in the order they appear in the results list. The return type is the type of the operand of `return` instructions, the type of `apply` instructions, and the type of the normal result of `try_apply` instructions.

  - An `@out` result is indirect. The address must be of an uninitialized object. The function is required to leave an initialized value there unless it terminates with a `throw` instruction or it has a non-Swift calling convention.
  - An `@owned` result is an owned direct result.
  - An `@autoreleased` result is an autoreleased direct result. If there is an autoreleased result, it must be the only direct result.
  - Otherwise, the parameter is an unowned direct result.

A direct parameter or result of trivial type must always be unowned.

An owned direct parameter or result is transferred to the recipient, which becomes responsible for destroying the value. This means that the value is passed at +1.

An unowned direct parameter or result is instantaneously valid at the point of transfer. The recipient does not need to worry about race conditions immediately destroying the value, but should copy it (e.g. by `strong_retain`ing an object pointer) if the value will be needed sooner rather than later.

A guaranteed direct parameter is like an unowned direct parameter value, except that it is guaranteed by the caller to remain valid throughout the execution of the call. This means that any `strong_retain`, `strong_release` pairs in the callee on the argument can be eliminated.

An autoreleased direct result must have a type with a retainable pointer representation. Autoreleased results are nominally transferred at +0, but the runtime takes steps to ensure that a +1 can be safely transferred, and those steps require precise code-layout control. Accordingly, the SIL pattern for an autoreleased convention looks exactly like the SIL pattern for an owned convention, and the extra runtime instrumentation is inserted on both sides when the SIL is lowered into LLVM IR. An autoreleased `apply` of a function that is defined with an autoreleased result has the effect of a +1 transfer of the result. An autoreleased `apply` of a function that is not defined with an autoreleased result has the effect of performing a strong retain in the caller. A non-autoreleased `apply` of a function that is defined with an autoreleased result has the effect of performing an autorelease in the callee.

- SIL function types may provide an optional error result, written by placing `@error` on a result. An error result is always

implicitly `@owned`. Only functions with a native calling convention may have an error result.

A function with an error result cannot be called with `apply`. It must be called with `try_apply`. There is one exception to this rule: a function with an error result can be called with `apply [nothrow]` if the compiler can prove that the function does not actually throw.

`return` produces a normal result of the function. To return an error result, use `throw`.

Type lowering lowers the `throws` annotation on formal function types into more concrete error propagation:

- For native Swift functions, `throws` is turned into an error result.
- For non-native Swift functions, `throws` is turned in an explicit error-handling mechanism based on the imported API. The importer only imports non-native methods and types as `throws` when it is possible to do this automatically.

- SIL function types may provide a pattern signature and substitutions to express that values of the type use a particular generic abstraction pattern. Both must be provided together. If a pattern signature is present, the component types (parameters, yields, and results) must be expressed in terms of the generic parameters of that signature. The pattern substitutions should be expressed in terms of the generic parameters of the overall generic signature, if any, or else the enclosing generic context, if any.

  A pattern signature follows the `@substituted` attribute, which must be the final attribute preceding the function type. Pattern substitutions follow the function type, preceded by the `for` keyword. For example:

  ```
  @substituted <T: Collection> (@in T) -> @out T.Element for Array<Int>
  ```

  The low-level representation of a value of this type may not match the representation of a value of the substituted-through version of it:

  ```
  (@in Array<Int>) -> @out Int
  ```

  Substitution differences at the outermost level of a function value may be adjusted using the `convert_function` instruction. Note that this only works at the outermost level and not in nested positions. For example, a function which takes a parameter of the first type above cannot be converted by `convert_function` to a function which takes a parameter of the second type; such a conversion must be done with a thunk.

  Type substitution on a function type with a pattern signature and substitutions only substitutes into the substitutions; the component types are preserved with their exact original structure.

- In the implementation, a SIL function type may also carry substitutions for its generic signature. This is a convenience for working with applied generic types and is not generally a formal part of the SIL language; in particular, values should not have such types. Such a type behaves like a non-generic type, as if the substitutions were actually applied to the underlying function type.

Async Functions

SIL function types may be `@async`. `@async` functions run inside async tasks, and can have explicit *suspend points* where they suspend execution. `@async` functions can only be called from other `@async` functions, but otherwise can be invoked with the normal `apply` and `try_apply` instructions (or `begin_apply` if they are coroutines).

In Swift, the `withUnsafeContinuation` primitive is used to implement primitive suspend points. In SIL, `@async` functions represent this abstraction using the `get_async_continuation[_addr]` and `await_async_continuation` instructions. `get_async_continuation[_addr]` accesses a *continuation* value that can be used to resume the coroutine after it suspends. The resulting continuation value can then be passed into a completion handler, registered with an event loop, or scheduled by some other mechanism. Operations on the continuation can resume the async function's execution by passing a value back to the async function, or passing in an error that propagates as an error in the async function's context. The `await_async_continuation` instruction suspends execution of the coroutine until the continuation is invoked to resume it. A use of `withUnsafeContinuation` in Swift:

```
func waitForCallback() async -> Int {
  return await withUnsafeContinuation { cc in
    registerCallback { cc.resume($0) }
  }
}
```

might lower to the following SIL:

```
sil @waitForCallback : $@convention(thin) @async () -> Int {
entry:
  %cc = get_async_continuation $Int
  %closure = function_ref @waitForCallback_closure
    : $@convention(thin) (UnsafeContinuation<Int>) -> ()
  apply %closure(%cc)
  await_async_continuation %cc, resume resume_cc

resume_cc(%result : $Int):
  return %result
}
```

The closure may then be inlined into the `waitForCallback` function:

```
sil @waitForCallback : $@convention(thin) @async () -> Int {
entry:
  %cc = get_async_continuation $Int
  %registerCallback = function_ref @registerCallback
    : $@convention(thin) (@convention(thick) () -> ()) -> ()
  %callback_fn = function_ref @waitForCallback_callback
  %callback = partial_apply %callback_fn(%cc)
  apply %registerCallback(%callback)
  await_async_continuation %cc, resume resume_cc

resume_cc(%result : $Int):
  return %result
}
```

Every continuation value must be used exactly once to resume its associated async coroutine once. It is undefined behavior to attempt to resume the same continuation more than once. On the flip side, failing to resume a continuation will leave the async task stuck in the suspended state, leaking any memory or other resources it owns.

Coroutine Types

A coroutine is a function which can suspend itself and return control to its caller without terminating the function. That is, it does not need to obey a strict stack discipline. SIL coroutines have control flow that is tightly integrated with their callers, and they pass information back and forth between caller and callee in a structured way through yield points. *Generalized accessors* and *generators* in Swift fit this description: a `read` or `modify` accessor coroutine projects a single value, yields ownership of that one value temporarily to the caller, and then takes ownership back when resumed, allowing the coroutine to clean up resources or otherwise react to mutations done by the caller. *Generators* similarly yield a stream of values one at a time to their caller, temporarily yielding ownership of each value in turn to the caller. The tight coupling of the caller's control flow with these coroutines allows the caller to *borrow* values produced by the coroutine, where a normal function return would need to transfer ownership of its return value, since a normal function's context ceases to exist and be able to maintain ownership of the value after it returns.

To support these concepts, SIL supports two kinds of coroutine: `@yield_many` and `@yield_once`. Either of these attributes may be written before a function type to indicate that it is a coroutine type. `@yield_many` and `@yield_once` coroutines are allowed to also be `@async`. (Note that `@async` functions are not themselves modeled explicitly as coroutines in SIL, although the implementation may use a coroutine lowering strategy.)

A coroutine type may declare any number of *yielded values*, which is to say, values which are provided to the caller at a yield point. Yielded values are written in the result list of a function type, prefixed by the `@yields` attribute. A yielded value may have a convention attribute, taken from the set of parameter attributes and interpreted as if the yield site were calling back to the calling function.

Currently, a coroutine may not have normal results.

Coroutine functions may be used in many of the same ways as normal function values. However, they cannot be called with the standard `apply` or `try_apply` instructions. A non-throwing yield-once coroutine can be called with the `begin_apply` instruction.

There is no support yet for calling a throwing yield-once coroutine or for calling a yield-many coroutine of any kind.

Coroutines may contain the special `yield` and `unwind` instructions.

A `@yield_many` coroutine may yield as many times as it desires. A `@yield_once` coroutine may yield exactly once before returning, although it may also `throw` before reaching that point.

Properties of Types

SIL classifies types into additional subgroups based on ABI stability and generic constraints:

- *Loadable types* are types with a fully exposed concrete representation:
  - Reference types
  - Builtin value types
  - Fragile struct types in which all element types are loadable
  - Tuple types in which all element types are loadable
  - Class protocol types
  - Archetypes constrained by a class protocol

  Values of loadable types are loaded and stored by loading and storing individual components of their representation. As a consequence:

  - values of loadable types can be loaded into SIL SSA values and stored from SSA values into memory without running any user-written code, although compiler-generated reference counting operations can happen.
  - values of loadable types can be take-initialized (moved between memory locations) with a bitwise copy.

  A *loadable aggregate type* is a tuple or struct type that is loadable.

  A *trivial type* is a loadable type with trivial value semantics. Values of trivial type can be loaded and stored without any retain or release operations and do not need to be destroyed.
- *Runtime-sized types* are restricted value types for which the compiler does not know the size of the type statically:
  - Resilient value types
  - Fragile struct or tuple types that contain resilient types as elements at any depth
  - Archetypes not constrained by a class protocol
- *Address-only types* are restricted value types which cannot be loaded or otherwise worked with as SSA values:
  - Runtime-sized types
  - Non-class protocol types
  - @weak types
  - Types that can't satisfy the requirements for being loadable because they care about the exact location of their value in memory and need to run some user-written code when they are copied or moved. Most commonly, types "care" about the addresses of values because addresses of values are registered in some global data structure, or because values may contain pointers into themselves. For example:
    - Addresses of values of Swift `@weak` types are registered in a global table. That table needs to be adjusted when a `@weak` value is copied or moved to a new address.
    - A non-COW collection type with a heap allocation (like `std::vector` in C++) needs to allocate memory and copy the collection elements when the collection is copied.
    - A non-COW string type that implements a small string optimization (like many implementations of `std::string` in C++) can contain a pointer into the value itself. That pointer needs to be recomputed when the string is copied or moved.

  Values of address-only type ("address-only values") must reside in memory and can only be referenced in SIL by address. Addresses of address-only values cannot be loaded from or stored to. SIL provides special instructions for indirectly manipulating address-only values, such as `copy_addr` and `destroy_addr`.

Some additional meaningful categories of type:

- A *heap object reference* type is a type whose representation consists of a single strong-reference-counted pointer. This includes all class types, the `Builtin.NativeObject` and `AnyObject` types, and archetypes that conform to one or more class protocols.
- A *reference type* is more general in that its low-level representation may include additional global pointers alongside a strong-reference-counted pointer. This includes all heap object reference types and adds thick function types and protocol/protocol composition types that conform to one or more class protocols. All reference types can be `retain`-ed and `release`-d. Reference types also have *ownership semantics* for their referenced heap object; see Reference Counting below.
- A type with *retainable pointer representation* is guaranteed to be compatible (in the C sense) with the Objective-C `id` type. The value at runtime may be `nil`. This includes classes, class metatypes, block functions, and class-bounded existentials with only Objective-C-compatible protocol constraints, as well as one level of `Optional` or `ImplicitlyUnwrappedOptional` applied to any of the above. Types with retainable pointer representation can be returned via the `@autoreleased` return convention.

SILGen does not always map Swift function types one-to-one to SIL function types. Function types are transformed in order to encode additional attributes:

- The **convention** of the function, indicated by the

  `@convention(`*convention*`)`

  attribute. This is similar to the language-level `@convention` attribute, though SIL extends the set of supported conventions with additional distinctions not exposed at the language level:
  - `@convention(thin)` indicates a "thin" function reference, which uses the Swift calling convention with no special "self" or "context" parameters.
  - `@convention(thick)` indicates a "thick" function reference, which uses the Swift calling convention and carries a reference-counted context object used to represent captures or other state required by the function. This attribute is implied by `@callee_owned` or `@callee_guaranteed`.
  - `@convention(block)` indicates an Objective-C compatible block reference. The function value is represented as a reference to the block object, which is an `id`-compatible Objective-C object that embeds its invocation function within the object. The invocation function uses the C calling convention.
  - `@convention(c)` indicates a C function reference. The function value carries no context and uses the C calling convention.
  - `@convention(objc_method)` indicates an Objective-C method implementation. The function uses the C calling convention, with the SIL-level `self` parameter (by SIL convention mapped to the final formal parameter) mapped to the `self` and `_cmd` arguments of the implementation.
  - `@convention(method)` indicates a Swift instance method implementation. The function uses the Swift calling convention, using the special `self` parameter.
  - `@convention(witness_method)` indicates a Swift protocol method implementation. The function's polymorphic convention is emitted in such a way as to guarantee that it is polymorphic across all possible implementors of the protocol.

Layout Compatible Types

(This section applies only to Swift 1.0 and will hopefully be obviated in future releases.)

SIL tries to be ignorant of the details of type layout, and low-level bit-banging operations such as pointer casts are generally undefined. However, as a concession to implementation convenience, some types are allowed to be considered **layout compatible**. Type `T` is *layout compatible* with type `U` iff:

- an address of type `$*U` can be cast by `address_to_pointer`/`pointer_to_address` to `$*T` and a valid value of type `T` can be loaded out (or indirectly used, if `T` is address- only),
- if `T` is a nontrivial type, then `retain_value`/`release_value` of the loaded `T` value is equivalent to `retain_value`/`release_value` of the original `U` value.

This is not always a commutative relationship; `T` can be layout-compatible with `U` whereas `U` is not layout-compatible with `T`. If the

layout compatible relationship does extend both ways, `T` and `U` are **commutatively layout compatible**. It is however always transitive; if `T` is layout-compatible with `U` and `U` is layout-compatible with `V`, then `T` is layout-compatible with `V`. All types are layout-compatible with themselves.

The following types are considered layout-compatible:

- `Builtin.RawPointer` is commutatively layout compatible with all heap object reference types, and `Optional` of heap object reference types. (Note that `RawPointer` is a trivial type, so does not have ownership semantics.)
- `Builtin.RawPointer` is commutatively layout compatible with `Builtin.Word`.
- Structs containing a single stored property are commutatively layout compatible with the type of that property.
- A heap object reference is commutatively layout compatible with any type that can correctly reference the heap object. For instance, given a class `B` and a derived class `D` inheriting from `B`, a value of type `B` referencing an instance of type `D` is layout compatible with both `B` and `D`, as well as `Builtin.NativeObject` and `AnyObject`. It is not layout compatible with an unrelated class type `E`.
- For payloaded enums, the payload type of the first payloaded case is layout-compatible with the enum (*not* commutatively).

Values and Operands

```
sil-identifier ::= [A-Za-z_0-9]+
sil-value-name ::= '%' sil-identifier
sil-value ::= sil-value-name
sil-value ::= 'undef'
sil-operand ::= sil-value ':' sil-type
```

SIL values are introduced with the `%` sigil and named by an alphanumeric identifier, which references the instruction or basic block argument that produces the value. SIL values may also refer to the keyword 'undef', which is a value of undefined contents.

Unlike LLVM IR, SIL instructions that take value operands *only* accept value operands. References to literal constants, functions, global variables, or other entities require specialized instructions such as `integer_literal`, `function_ref`, `global_addr`, etc.

Functions

```
decl ::= sil-function
sil-function ::= 'sil' sil-linkage? sil-function-attribute+
                 sil-function-name ':' sil-type
                 '{' sil-basic-block+ '}'
sil-function-name ::= '@' [A-Za-z_0-9]+
```

SIL functions are defined with the `sil` keyword. SIL function names are introduced with the `@` sigil and named by an alphanumeric identifier. This name will become the LLVM IR name for the function, and is usually the mangled name of the originating Swift declaration. The `sil` syntax declares the function's name and SIL type, and defines the body of the function inside braces. The declared type must be a function type, which may be generic.

Function Attributes

```
sil-function-attribute ::= '[canonical]'
```

The function is in canonical SIL even if the module is still in raw SIL.

```
sil-function-attribute ::= '[ossa]'
```

The function is in OSSA (ownership SSA) form.

```
sil-function-attribute ::= '[transparent]'
```

Transparent functions are always inlined and don't keep their source information when inlined.

```
sil-function-attribute ::= '[' sil-function-thunk ']'
sil-function-thunk ::= 'thunk'
sil-function-thunk ::= 'signature_optimized_thunk'
sil-function-thunk ::= 'reabstraction_thunk'
```

The function is a compiler generated thunk.

```
sil-function-attribute ::= '[dynamically_replacable]'
```

The function can be replaced at runtime with a different implementation. Optimizations must not assume anything about such a function, even if the SIL of the function body is available.

```
sil-function-attribute ::= '[dynamic_replacement_for' identifier ']'
sil-function-attribute ::= '[objc_replacement_for' identifier ']'
```

Specifies for which function this function is a replacement.

```
sil-function-attribute ::= '[exact_self_class]'
```

The function is a designated initializers, where it is known that the static type being allocated is the type of the class that defines the designated initializer.

```
sil-function-attribute ::= '[without_actually_escaping]'
```

The function is a thunk for closures which are not actually escaping.

```
sil-function-attribute ::= '[' sil-function-purpose ']'
sil-function-purpose ::= 'global_init'
```

The implied semantics are:

- side-effects can occur any time before the first invocation.
- all calls to the same `global_init` function have the same side-effects.
- any operation that may observe the initializer's side-effects must be preceded by a call to the initializer.

This is currently true if the function is an addressor that was lazily generated from a global variable access. Note that the initialization function itself does not need this attribute. It is private and only called within the addressor.

```
sil-function-purpose ::= 'lazy_getter'
```

The function is a getter of a lazy property for which the backing storage is an `Optional` of the property's type. The getter contains a top-level switch_enum (or switch_enum_addr), which tests if the lazy property is already computed. In the `None`-case, the property is computed and stored to the backing storage of the property.

After the first call of a lazy property getter, it is guaranteed that the property is computed and consecutive calls always execute the `Some`-case of the top-level switch_enum.

```
sil-function-attribute ::= '[weak_imported]'
```

Cross-module references to this function should always use weak linking.

```
sil-function-attribute ::= '[available' sil-version-tuple ']'
sil-version-tuple ::= [0-9]+ ('.' [0-9]+)*
```

The minimal OS-version where the function is available.

```
sil-function-attribute ::= '[' sil-function-inlining ']'
sil-function-inlining ::= 'never'
```

The function is never inlined.

```
sil-function-inlining ::= 'always'
```

The function is always inlined, even in a `Onone` build.

```
sil-function-attribute ::= '[' sil-function-optimization ']'
sil-function-inlining ::= 'Onone'
sil-function-inlining ::= 'Ospeed'
```

```
sil-function-inlining ::= 'Osize'
```

The function is optimized according to this attribute, overriding the setting from the command line.

```
sil-function-attribute ::= '[' sil-function-effects ']'
sil-function-effects ::= 'readonly'
sil-function-effects ::= 'readnone'
sil-function-effects ::= 'readwrite'
sil-function-effects ::= 'releasenone'
```

The specified memory effects of the function.

```
sil-function-attribute ::= '[' 'escapes' escape-list ']'
sil-function-attribute ::= '[' 'defined_escapes' escape-list ']'
escape-list ::= (escape-list ',')? escape
escape ::= '!' arg-selection                // not-escaping
escape ::= arg-selection '=>' arg-selection  // exclusive escaping
escape ::= arg-selection '->' arg-selection  // not-exclusive escaping
arg-selection ::= arg-or-return ('.' projection-path)?
arg-or-return ::= '%' [0-9]+
arg-or-return ::= '%r'
projection-path ::= (projection-path '.')? path-component
path-component ::= 's' [0-9]+        // struct field
path-component ::= 'c' [0-9]+        // class field
path-component ::= 'ct'              // class tail element
path-component ::= 'e' [0-9]+        // enum case
path-component ::= [0-9]+            // tuple element
path-component ::= 'v**'             // any value fields
path-component ::= 'c*'              // any class field
path-component ::= '**'              // anything
```

The escaping effects for function arguments. For details see the documentation in `SwiftCompilerSources/Sources/SIL/Effects.swift`.

```
sil-function-attribute ::= '[_semantics "' [A-Za-z._0-9]+ '"]'
```

The specified high-level semantics of the function. The optimizer can use this information to perform high-level optimizations before such functions are inlined. For example, `Array` operations are annotated with semantic attributes to let the optimizer perform redundant bounds check elimination and similar optimizations.

```
sil-function-attribute ::= '[_specialize "' [A-Za-z._0-9]+ '"]'
```

Specifies for which types specialized code should be generated.

```
sil-function-attribute ::= '[clang "' identifier '"]'
```

The clang node owner.

```
sil-function-attribute ::= '[' performance-constraint ']'
performance-constraint :: 'no_locks'
performance-constraint :: 'no_allocation'
```

Specifies the performance constraints for the function, which defines which type of runtime functions are allowed to be called from the function.

Basic Blocks

```
sil-basic-block ::= sil-label sil-instruction-def* sil-terminator
sil-label ::= sil-identifier ('(' sil-argument (',' sil-argument)* ')')? ':'
sil-value-ownership-kind ::= @owned
sil-value-ownership-kind ::= @guaranteed
sil-value-ownership-kind ::= @unowned
sil-argument ::= sil-value-name ':' sil-value-ownership-kind? sil-type

sil-instruction-result ::= sil-value-name
sil-instruction-result ::= '(' (sil-value-name (',' sil-value-name)*)? ')'
sil-instruction-source-info ::= (',' sil-scope-ref)? (',' sil-loc)?
sil-instruction-def ::=
  (sil-instruction-result '=')? sil-instruction sil-instruction-source-info
```

A function body consists of one or more basic blocks that correspond to the nodes of the function's control flow graph. Each basic block contains one or more instructions and ends with a terminator instruction. The function's entry point is always the first basic block in its body.

In SIL, basic blocks take arguments, which are used as an alternative to LLVM's phi nodes. Basic block arguments are bound by the branch from the predecessor block:

```
sil @iif : $(Builtin.Int1, Builtin.Int64, Builtin.Int64) -> Builtin.Int64 {
bb0(%cond : $Builtin.Int1, %ifTrue : $Builtin.Int64, %ifFalse : $Builtin.Int64):
  cond_br %cond : $Builtin.Int1, then, else
then:
  br finish(%ifTrue : $Builtin.Int64)
else:
  br finish(%ifFalse : $Builtin.Int64)
finish(%result : $Builtin.Int64):
  return %result : $Builtin.Int64
}
```

Arguments to the entry point basic block, which has no predecessor, are bound by the function's caller:

```
sil @foo : $@convention(thin) (Int) -> Int {
bb0(%x : $Int):
  return %x : $Int
}

sil @bar : $@convention(thin) (Int, Int) -> () {
bb0(%x : $Int, %y : $Int):
  %foo = function_ref @foo
  %1 = apply %foo(%x) : $(Int) -> Int
  %2 = apply %foo(%y) : $(Int) -> Int
  %3 = tuple ()
  return %3 : $()
}
```

When a function is in Ownership SSA, arguments additionally have an explicit annotated convention that describe the ownership semantics of the argument value:

```
sil [ossa] @baz : $@convention(thin) (Int, @owned String, @guaranteed String, @unowned String) -> () {
bb0(%x : $Int, %y : @owned $String, %z : @guaranteed $String, %w : @unowned $String):
  ...
}
```

Note that the first argument (`%x`) has an implicit ownership kind of `@none` since all trivial values have `@none` ownership.

Debug Information

```
sil-scope-ref ::= 'scope' [0-9]+
sil-scope ::= 'sil_scope' [0-9]+ '{'
                sil-loc
                'parent' scope-parent
                ('inlined_at' sil-scope-ref)?
              '}'
scope-parent ::= sil-function-name ':' sil-type
scope-parent ::= sil-scope-ref
sil-loc ::= 'loc' string-literal ':' [0-9]+ ':' [0-9]+
```

Each instruction may have a debug location and a SIL scope reference at the end. Debug locations consist of a filename, a line number, and a column number. If the debug location is omitted, it defaults to the location in the SIL source file. SIL scopes describe

the position inside the lexical scope structure that the Swift expression a SIL instruction was generated from had originally. SIL scopes also hold inlining information.

Declaration References

```
sil-decl-ref ::= '#' sil-identifier ('.' sil-identifier)* sil-decl-subref?
sil-decl-subref ::= '!' sil-decl-subref-part ('.' sil-decl-lang)? ('.' sil-decl-autodiff)?
sil-decl-subref ::= '!' sil-decl-lang
sil-decl-subref ::= '!' sil-decl-autodiff
sil-decl-subref-part ::= 'getter'
sil-decl-subref-part ::= 'setter'
sil-decl-subref-part ::= 'allocator'
sil-decl-subref-part ::= 'initializer'
sil-decl-subref-part ::= 'enumelt'
sil-decl-subref-part ::= 'destroyer'
sil-decl-subref-part ::= 'deallocator'
sil-decl-subref-part ::= 'globalaccessor'
sil-decl-subref-part ::= 'ivardestroyer'
sil-decl-subref-part ::= 'ivarinitializer'
sil-decl-subref-part ::= 'defaultarg' '.' [0-9]+
sil-decl-lang ::= 'foreign'
sil-decl-autodiff ::= sil-decl-autodiff-kind '.' sil-decl-autodiff-indices
sil-decl-autodiff-kind ::= 'jvp'
sil-decl-autodiff-kind ::= 'vjp'
sil-decl-autodiff-indices ::= [SU]+
```

Some SIL instructions need to reference Swift declarations directly. These references are introduced with the `#` sigil followed by the fully qualified name of the Swift declaration. Some Swift declarations are decomposed into multiple entities at the SIL level. These are distinguished by following the qualified name with `!` and one or more `.`-separated component entity discriminators:

- `getter`: the getter function for a `var` declaration
- `setter`: the setter function for a `var` declaration
- `allocator`: a `struct` or `enum` constructor, or a `class`'s *allocating constructor*
- `initializer`: a `class`'s *initializing constructor*
- `enumelt`: a member of a `enum` type.
- `destroyer`: a class's destroying destructor
- `deallocator`: a class's deallocating destructor
- `globalaccessor`: the addressor function for a global variable
- `ivardestroyer`: a class's ivar destroyer
- `ivarinitializer`: a class's ivar initializer
- `defaultarg.`*n*: the default argument-generating function for the *n*-th argument of a Swiff `func`
- `foreign`: a specific entry point for C/Objective-C interoperability

Linkage

```
sil-linkage ::= 'public'
sil-linkage ::= 'hidden'
sil-linkage ::= 'shared'
sil-linkage ::= 'private'
sil-linkage ::= 'public_external'
sil-linkage ::= 'hidden_external'
sil-linkage ::= 'non_abi'
```

A linkage specifier controls the situations in which two objects in different SIL modules are *linked*, i.e. treated as the same object.

A linkage is *external* if it ends with the suffix `external`. An object must be a definition if its linkage is not external.

All functions, global variables, and witness tables have linkage. The default linkage of a definition is `public`. The default linkage of a declaration is `public_external`. (These may eventually change to `hidden` and `hidden_external`, respectively.)

On a global variable, an external linkage is what indicates that the variable is not a definition. A variable lacking an explicit linkage specifier is presumed a definition (and thus gets the default linkage for definitions, `public`.)

Definition of the *linked* relation

Two objects are linked if they have the same name and are mutually visible:

- An object with `public` or `public_external` linkage is always visible.
- An object with `hidden`, `hidden_external`, or `shared` linkage is visible only to objects in the same Swift module.
- An object with `private` linkage is visible only to objects in the same SIL module.

Note that the *linked* relationship is an equivalence relation: it is reflexive, symmetric, and transitive.

Requirements on linked objects

If two objects are linked, they must have the same type.

If two objects are linked, they must have the same linkage, except:

- A `public` object may be linked to a `public_external` object.
- A `hidden` object may be linked to a `hidden_external` object.

If two objects are linked, at most one may be a definition, unless:

- both objects have `shared` linkage or
- at least one of the objects has an external linkage.

If two objects are linked, and both are definitions, then the definitions must be semantically equivalent. This equivalence may exist only on the level of user-visible semantics of well-defined code; it should not be taken to guarantee that the linked definitions are exactly operationally equivalent. For example, one definition of a function might copy a value out of an address parameter, while another may have had an analysis applied to prove that said value is not needed.

If an object has any uses, then it must be linked to a definition with non-external linkage.

Public non-ABI linkage

The *non_abi* linkage is a special linkage used for definitions which only exist in serialized SIL, and do not define visible symbols in the object file.

A definition with *non_abi* linkage behaves like it has *shared* linkage, except that it must be serialized in the SIL module even if not referenced from anywhere else in the module. For example, this means it is considered a root for dead function elimination.

When a *non_abi* definition is deserialized, it will have *shared_external* linkage.

There is no *non_abi_external* linkage. Instead, when referencing a *non_abi* declaration that is defined in a different translation unit from the same Swift module, you must use *hidden_external* linkage.

Summary

- `public` definitions are unique and visible everywhere in the program. In LLVM IR, they will be emitted with `external` linkage and `default` visibility.
- `hidden` definitions are unique and visible only within the current Swift module. In LLVM IR, they will be emitted with `external` linkage and `hidden` visibility.
- `private` definitions are unique and visible only within the current SIL module. In LLVM IR, they will be emitted with `private` linkage.
- `shared` definitions are visible only within the current Swift module. They can be linked only with other `shared` definitions, which must be equivalent; therefore, they only need to be emitted if actually used. In LLVM IR, they will be emitted with `linkonce_odr` linkage and `hidden` visibility.
- `public_external` and `hidden_external` objects always have visible definitions somewhere else. If this object

nonetheless has a definition, it's only for the benefit of optimization or analysis. In LLVM IR, declarations will have `external` linkage and definitions (if actually emitted as definitions) will have `available_externally` linkage.

VTables

```
decl ::= sil-vtable
sil-vtable ::= 'sil_vtable' identifier '{' sil-vtable-entry* '}'

sil-vtable-entry ::= sil-decl-ref ':' sil-linkage? sil-function-name
```

SIL represents dynamic dispatch for class methods using the class_method, super_method, objc_method, and objc_super_method instructions.

The potential destinations for class_method and super_method are tracked in `sil_vtable` declarations for every class type. The declaration contains a mapping from every method of the class (including those inherited from its base class) to the SIL function that implements the method for that class:

```
class A {
  func foo()
  func bar()
  func bas()
}

sil @A_foo : $@convention(thin) (@owned A) -> ()
sil @A_bar : $@convention(thin) (@owned A) -> ()
sil @A_bas : $@convention(thin) (@owned A) -> ()

sil_vtable A {
  #A.foo: @A_foo
  #A.bar: @A_bar
  #A.bas: @A_bas
}

class B : A {
  func bar()
}

sil @B_bar : $@convention(thin) (@owned B) -> ()

sil_vtable B {
  #A.foo: @A_foo
  #A.bar: @B_bar
  #A.bas: @A_bas
}

class C : B {
  func bas()
}

sil @C_bas : $@convention(thin) (@owned C) -> ()

sil_vtable C {
  #A.foo: @A_foo
  #A.bar: @B_bar
  #A.bas: @C_bas
}
```

Note that the declaration reference in the vtable is to the least-derived method visible through that class (in the example above, `B`'s vtable references `A.bar` and not `B.bar`, and `C`'s vtable references `A.bas` and not `C.bas`). The Swift AST maintains override relationships between declarations that can be used to look up overridden methods in the SIL vtable for a derived class (such as `C.bas` in `C`'s vtable).

In case the SIL function is a thunk, the function name is preceded with the linkage of the original implementing function.

Witness Tables

```
decl ::= sil-witness-table
sil-witness-table ::= 'sil_witness_table' sil-linkage?
                      normal-protocol-conformance '{' sil-witness-entry* '}'
```

SIL encodes the information needed for dynamic dispatch of generic types into witness tables. This information is used to produce runtime dispatch tables when generating binary code. It can also be used by SIL optimizations to specialize generic functions. A witness table is emitted for every declared explicit conformance. Generic types share one generic witness table for all of their instances. Derived classes inherit the witness tables of their base class.

```
protocol-conformance ::= normal-protocol-conformance
protocol-conformance ::= 'inherit' '(' protocol-conformance ')'
protocol-conformance ::= 'specialize' '<' substitution* '>'
                         '(' protocol-conformance ')'
protocol-conformance ::= 'dependent'
normal-protocol-conformance ::= identifier ':' identifier 'module' identifier
```

Witness tables are keyed by *protocol conformance*, which is a unique identifier for a concrete type's conformance to a protocol.

- A *normal protocol conformance* names a (potentially unbound generic) type, the protocol it conforms to, and the module in which the type or extension declaration that provides the conformance appears. These correspond 1:1 to protocol conformance declarations in the source code.
- If a derived class conforms to a protocol through inheritance from its base class, this is represented by an *inherited protocol conformance*, which simply references the protocol conformance for the base class.
- If an instance of a generic type conforms to a protocol, it does so with a *specialized conformance*, which provides the generic parameter bindings to the normal conformance, which should be for a generic type.

Witness tables are only directly associated with normal conformances. Inherited and specialized conformances indirectly reference the witness table of the underlying normal conformance.

```
sil-witness-entry ::= 'base_protocol' identifier ':' protocol-conformance
sil-witness-entry ::= 'method' sil-decl-ref ':' sil-function-name
sil-witness-entry ::= 'associated_type' identifier
sil-witness-entry ::= 'associated_type_protocol'
                      '(' identifier ':' identifier ')' ':' protocol-conformance
```

Witness tables consist of the following entries:

- *Base protocol entries* provide references to the protocol conformances that satisfy the witnessed protocols' inherited protocols.
- *Method entries* map a method requirement of the protocol to a SIL function that implements that method for the witness type. One method entry must exist for every required method of the witnessed protocol.
- *Associated type entries* map an associated type requirement of the protocol to the type that satisfies that requirement for the witness type. Note that the witness type is a source-level Swift type and not a SIL type. One associated type entry must exist for every required associated type of the witnessed protocol.
- *Associated type protocol entries* map a protocol requirement on an associated type to the protocol conformance that satisfies that requirement for the associated type.

Default Witness Tables

```
decl ::= sil-default-witness-table
sil-default-witness-table ::= 'sil_default_witness_table'
                              identifier minimum-witness-table-size
                              '{' sil-default-witness-entry* '}'
minimum-witness-table-size ::= integer
```

SIL encodes requirements with resilient default implementations in a default witness table. We say a requirement has a resilient default implementation if the following conditions hold:

- The requirement has a default implementation
- The requirement is either the last requirement in the protocol, or all subsequent requirements also have resilient default implementations

The set of requirements with resilient default implementations is stored in protocol metadata.

The minimum witness table size is the size of the witness table, in words, not including any requirements with resilient default implementations.

Any conforming witness table must have a size between the minimum size, and the maximum size, which is equal to the minimum size plus the number of default requirements.

At load time, if the runtime encounters a witness table with fewer than the maximum number of witnesses, the witness table is copied, with default witnesses copied in. This ensures that callers can always expect to find the correct number of requirements in each witness table, and new requirements can be added by the framework author, without breaking client code, as long as the new requirements have resilient default implementations.

Default witness tables are keyed by the protocol itself. Only protocols with public visibility need a default witness table; private and internal protocols are never seen outside the module, therefore there are no resilience issues with adding new requirements.

```
sil-default-witness-entry ::= 'method' sil-decl-ref ':' sil-function-name
```

Default witness tables currently contain only one type of entry:

- *Method entries* map a method requirement of the protocol to a SIL function that implements that method in a manner suitable for all witness types.

Global Variables

```
decl ::= sil-global-variable
static-initializer ::= '=' '{' sil-instruction-def* '}'
sil-global-variable ::= 'sil_global' sil-linkage identifier ':' sil-type
                           (static-initializer)?
```

SIL representation of a global variable.

Global variable access is performed by the `alloc_global`, `global_addr` and `global_value` instructions.

A global can have a static initializer if its initial value can be composed of literals. The static initializer is represented as a list of literal and aggregate instructions where the last instruction is the top-level value of the static initializer:

```
sil_global hidden @$S4test3varSiv : $Int {
  %0 = integer_literal $Builtin.Int64, 27
  %initval = struct $Int (%0 : $Builtin.Int64)
}
```

If a global does not have a static initializer, the `alloc_global` instruction must be performed prior an access to initialize the storage. Once a global's storage has been initialized, `global_addr` is used to project the value.

If the last instruction in the static initializer is an `object` instruction the global variable is a statically initialized object. In this case the variable cannot be used as l-value, i.e. the reference to the object cannot be modified. As a consequence the variable cannot be accessed with `global_addr` but only with `global_value`.

Differentiability Witnesses

```
decl ::= sil-differentiability-witness
sil-differentiability-witness ::=
    'sil_differentiability_witness'
    sil-linkage?
    '[' differentiability-kind ']'
    '[' 'parameters' sil-differentiability-witness-function-index-list ']'
    '[' 'results' sil-differentiability-witness-function-index-list ']'
    generic-parameter-clause?
    sil-function-name ':' sil-type
    sil-differentiability-witness-body?

differentiability-kind ::= 'forward' | 'reverse' | 'normal' | 'linear'

sil-differentiability-witness-body ::=
    '{' sil-differentiability-witness-entry?
        sil-differentiability-witness-entry? '}'

sil-differentiability-witness-entry ::=
    sil-differentiability-witness-entry-kind ':'
    sil-entry-name ':' sil-type

sil-differentiability-witness-entry-kind ::= 'jvp' | 'vjp'
```

SIL encodes function differentiability via differentiability witnesses.

Differentiability witnesses map a "key" (including an "original" SIL function) to derivative SIL functions.

Differentiability witnesses are keyed by the following:

- An "original" SIL function name.
- Differentiability parameter indices.
- Differentiability result indices.
- A generic parameter clause, representing differentiability generic requirements.

Differentiability witnesses may have a body, specifying derivative functions for the key. Verification checks that derivative functions have the expected type based on the key.

```
sil_differentiability_witness hidden [normal] [parameters 0] [results 0] <T where T : Differentiable> @id : $@convention(thin) (T) -> T {
  jvp: @id_jvp : $@convention(thin) (T) -> (T, @owned @callee_guaranteed (T.TangentVector) -> T.TangentVector)
  vjp: @id_vjp : $@convention(thin) (T) -> (T, @owned @callee_guaranteed (T.TangentVector) -> T.TangentVector)
}
```

During SILGen, differentiability witnesses are emitted for the following:

- *@differentiable* declaration attributes.
- *@derivative* declaration attributes. Registered derivative functions become differentiability witness entries.

The SIL differentiation transform canonicalizes differentiability witnesses, filling in missing entries.

Differentiability witness entries are accessed via the *differentiability_witness_function* instruction.

Dataflow Errors

Dataflow errors may exist in raw SIL. Swift's semantics defines these conditions as errors, so they must be diagnosed by diagnostic passes and must not exist in canonical SIL.

Definitive Initialization

Swift requires that all local variables be initialized before use. In constructors, all instance variables of a struct, enum, or class type must be initialized before the object is used and before the constructor is returned from.

Unreachable Control Flow

The `unreachable` terminator is emitted in raw SIL to mark incorrect control flow, such as a non-`Void` function failing to `return` a value, or a `switch` statement failing to cover all possible values of its subject. The guaranteed dead code elimination pass can eliminate truly unreachable basic blocks, or `unreachable` instructions may be dominated by applications of functions returning uninhabited types. An `unreachable` instruction that survives guaranteed DCE and is not immediately preceded by a no-return application is a dataflow error.

Ownership SSA

A SILFunction marked with the `[ossa]` function attribute is considered to be in Ownership SSA form. Ownership SSA is an augmented version of SSA that enforces ownership invariants by imbuing value-operand edges with semantic ownership information. All SIL values are assigned a constant ownership kind that defines the ownership semantics that the value models. All SIL operands that use a SIL value are required to be able to be semantically partitioned in between "non-lifetime ending uses" that just require the value to be live and "lifetime ending uses" that end the lifetime of the value and after which the value can no longer be used. Since by definition operands that are lifetime ending uses end their associated value's lifetime, we must have that, ignoring program ending Dead End Blocks, the lifetime ending use points jointly post-dominate all non-lifetime ending use points and that a value must have exactly one lifetime ending use along all reachable program paths, preventing leaks and use-after-frees. As an example, consider the following SIL example with partitioned defs/uses annotated inline:

```
sil @stash_and_cast : $@convention(thin) (@owned Klass) -> @owned SuperKlass {
bb0(%kls1 : @owned $Klass): // Definition of %kls1

  // "Normal Use" kls1.
  // Definition of %kls2.
  %kls2 = copy_value %kls1 : $Klass

  // "Consuming Use" of %kls2 to store it into a global. Stores in ossa are
  // consuming since memory is generally assumed to have "owned"
  // semantics. After this instruction executes, we can no longer use %kls2
  // without triggering an ownership violation.
  store %kls2 to [init] %globalMem : $*Klass

  // "Consuming Use" of %kls1.
  // Definition of %kls1Casted.
  %kls1Casted = upcast %kls1 : $Klass to $SuperKlass

  // "Consuming Use" of %kls1Casted
  return %kls1Casted : $SuperKlass
}
```

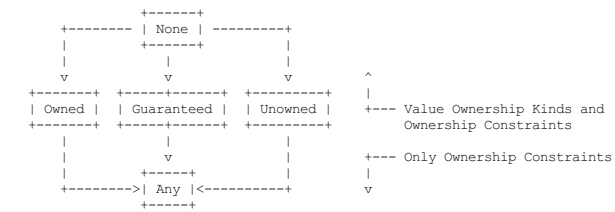
Notice how every value in the SIL above has a partionable set of uses with normal uses always before consuming uses. Any such violations of ownership semantics would trigger a SILVerifier error allowing us to know that we do not have any leaks or use-after-frees in the above code.

Ownership Kind

The semantics in the previous example is of just one form of ownership semantics supported: "owned" semantics. In SIL, we map these "ownership semantics" into a form that a compiler can reason about by mapping semantics onto a lattice with the following elements: None, Owned, Guaranteed, Unowned, *Any*. We call this the lattice of "Ownership Kinds" and each individual value an "Ownership Kind". This lattice is defined as a 3-level lattice with:

```
1. None being Top.
2. Any being Bottom.
3. All non-Any, non-None OwnershipKinds being defined as a mid-level elements of the lattice
```

We can graphically represent the lattice via a diagram like the following:



One moves down the lattice by performing a "meet" operation:

```
None meet OtherOwnershipKind -> OtherOwnershipKind
Unowned meet Owned -> Any
Owned meet Guaranteed -> Any
```

and one moves up the lattice by performing a "join" operation, e.x.:

```
Any join OtherOwnershipKind -> OtherOwnershipKind
Owned join Any -> Owned
Owned join Guaranteed -> None
```

This lattice is applied to SIL by requiring well formed SIL to:

1. Define a map of each SIL value to a constant OwnershipKind that classify the semantics that the SIL value obeys. This ownership kind may be static (i.e.: the same for all instances of an instruction) or dynamic (e.x.: forwarding instructions set their ownership upon construction). We call this subset of OwnershipKind to be the set of Value Ownership Kind: None, Unowned, Guaranteed, Owned (note conspiciously missing *Any*). This is because in our model *Any* represents an unknown ownership semantics and since our model is strict, we do not allow for values to have unknown ownership.
2. Define a map from each operand of a SILInstruction, *i*, to a constant Ownership Kind, Boolean pair called the operand's Ownership Constraint. The Ownership Kind element of the Ownership Constraint determines semantically which ownership kind's the operand's value can take on. The Boolean value is used to know if an operand will end the lifetime of the incoming value when checking dataflow rules. The dataflow rules that each Value Ownership Kind obeys is documented for each Value Ownership Kind in its detailed description below.

Then we take these two maps and require that valid SIL has the property that given an operand, `op(i)` of an instruction `i` and a value `v` that `op(i)` can only use `v` if the `join` of `OwnershipConstraint(operand(i))` with `ValueOwnershipKind(v)` is equal to the `ValueOwnershipKind` of `v`. In symbols, we must have that:

```
join : (OwnershipConstraint, ValueOwnershipKind) -> ValueOwnershipKind
OwnershipConstraint(operand(i)) join ValueOwnershipKind(v) = ValueOwnershipKind(v)
```

In words, a value can be passed to an operand if applying the operand's ownership constraint to the value's ownership does not change the value's ownership. Operationally this has a few interesting effects on SIL:

1. We have defined away invalid value-operand (aka def-use) pairing since the SILVerifier validates the aforementioned relationship on all SIL values, uses at all points of the pipeline until ossa is lowered.
2. Many SIL instructions do not care about the ownership kind that their value will take. They can just define all of their operand's as having an ownership constraint of Any.

Now lets go into more depth upon Value Ownership Kind and Ownership Constraint.

Value Ownership Kind

As mentioned above, each SIL value is statically mapped to an Ownership Kind called the value's "ValueOwnershipKind" that classify the semantics of the value. Below, we map each ValueOwnershipKind to a short summary of the semantics implied upon the parent value:

- **None**. This is used to represent values that do not require memory management and are outside of Ownership SSA invariants. Examples: trivial values (e.x.: Int, Float), non-payloaded cases of non-trivial enums (e.x.: Optional<T>.none), all address types.
- **Owned**. A value that exists independently of any other value and is consumed exactly once along all paths through a function by either a destroy_value (actually destroying the value) or by a consuming instruction that rebinds the value in some manner (e.x.: apply, casts, store).
- **Guaranteed**. A value with a scoped lifetime whose liveness is dependent on the lifetime of some other "base" owned or guaranteed value. Consumed by instructions like end_borrow. The "base" value is statically guaranteed to be live at all of the value's paired end_borrow instructions.
- **Unowned**. A value that is only guaranteed to be instantaneously valid and must be copied before the value is used in an

`@owned` or `@guaranteed` context. This is needed both to model argument values with the ObjC unsafe unowned argument convention and also to model the ownership resulting from bitcasting a trivial type to a non-trivial type. This value should never be consumed.

We describe each of these semantics in below in more detail.

Owned

Owned ownership models "move only" values. We require that each such value is consumed exactly once along all program paths. The IR verifier will flag values that are not consumed along a path as a leak and any double consumes as use-after-frees. We model move operations via forwarding uses such as casts and transforming terminators (e.x.: switch_enum, checked_cast_br) that transform the input value, consuming it in the process, and producing a new transformed owned value as a result.

Putting this all together, one can view each owned SIL value as being effectively a "move only value" except when explicitly copied by a copy_value. This of course implies that ARC operations can be assumed to only semantically effect the specific value that they are applied to /and/ that each ARC constraint is able to be verified independently for each owned SILValue derived from the ARC object. As an example, consider the following Swift/SIL:

```
// testcase.swift.
func doSomething(x : Klass) -> OtherKlass? {
  return x as? OtherKlass
}

// testcase.sil. A possible SILGen lowering
sil [ossa] @doSomething : $@convention(thin) (@guaranteed Klass) -> () {
bb0(%0 : @guaranteed Klass):
  // Definition of '%1'
  %1 = copy_value %0 : $Klass

  // Consume '%1'. This means '%1' can no longer be used after this point. We
  // rebind '%1' in the destination blocks (bbYes, bbNo).
  checked_cast_br %1 : $Klass to $OtherKlass, bbYes, bbNo

bbYes(%2 : @owned $OtherKlass): // On success, the checked_cast_br forwards
                                // '%1' into '%2' after casting to OtherKlass.

  // Forward '%2' into '%3'. '%2' can not be used past this point in the
  // function.
  %3 = enum $Optional<OtherKlass>, case #Optional.some!enumelt, %2 : $OtherKlass

  // Forward '%3' into the branch. '%3' can not be used past this point.
  br bbEpilog(%3 : $Optional<OtherKlass>)

bbNo(%3 : @owned $Klass): // On failure, since we consumed '%1' already, we
                          // return the original '%1' as a new value '%3'
                          // so we can use it below.
  // Actually destroy the underlying copy (``%1``) created by the copy_value
  // in bb0.
  destroy_value %3 : $Klass

  // We want to return nil here. So we create a new non-payloaded enum and
  // pass it off to bbEpilog.
  %4 = enum $Optional<OtherKlass>, case #Optional.none!enumelt
  br bbEpilog(%4 : $Optional<OtherKlass>)

bbEpilog(%5 : @owned $Optional<OtherKlass>):
  // Consumes '%5' to return to caller.
  return %5 : $Optional<OtherKlass>
}
```

Notice how our individual copy (`%1`) threads its way through the IR using forwarding uses of `@owned` ownership. These forwarding uses partition the lifetime of the result of the copy_value into a set of disjoint individual owned lifetimes (`%2`, `%3`, `%5`).

Guaranteed

Guaranteed ownership models values that have a scoped dependent lifetime on a "base value" with owned or guaranteed ownership. Due to this lifetime dependence, the base value is required to be statically live over the entire scope where the guaranteed value is valid.

These explicit scopes are introduced into SIL by begin scope instructions (e.x.: begin_borrow, load_borrow) that are paired with sets of jointly post-dominating scope ending instructions (e.x.: end_borrow):

```
sil [ossa] @guaranteed_values : $@convention(thin) (@owned Klass) -> () {
bb0(%0 : @owned $Klass):
  %1 = begin_borrow %0 : $Klass
  cond_br ..., bb1, bb2

bb1:
  ...
  end_borrow %1 : $Klass
  destroy_value %0 : $Klass
  br bb3

bb2:
  ...
  end_borrow %1 : $Klass
  destroy_value %0 : $Klass
  br bb3

bb3:
  ...
}
```

Notice how the end_borrow allow for a SIL generator to communicate to optimizations that they can never shrink the lifetime of `%0` by moving destroy_value above `%1`.

Values with guaranteed ownership follow a dataflow rule that states that non-consuming forwarding uses of the guaranteed value are also guaranteed and are recursively validated as being in the original values scope. This was a choice we made to reduce idempotent scopes in the IR:

```
sil [ossa] @get_first_elt : $@convention(thin) (@guaranteed (String, String)) -> @owned String {
bb0(%0 : @guaranteed $(String, String)):
  // %1 is validated as if it was apart of %0 and does not need its own begin_borrow/end_borrow.
  %1 = tuple_extract %0 : $(String, String)
  // So this copy_value is treated as a use of %0.
  %2 = copy_value %1 : $String
  return %2 : $String
}
```

None

Values with None ownership are inert values that exist outside of the guarantees of Ownership SSA. Some examples of such values are:

- Trivially typed values such as: Int, Float, Double
- Non-payloaded non-trivial enums.
- Address types.

Since values with none ownership exist outside of ownership SSA, they can be used like normal SSA without violating ownership SSA invariants. This does not mean that code does not potentially violate other SIL rules (consider memory lifetime invariants):

```
sil @none_values : $@convention(thin) (Int, @in Klass) -> Int {
bb0(%0 : $Int, %1 : $*Klass):

  // %0, %1 are normal SSA values that can be used anywhere in the function
  // without breaking Ownership SSA invariants. It could violate other
  // invariants if for instance, we load from %1 after we destroy the object
```

```
  // there.
  destroy_addr %1 : $*Klass

  // If uncommented, this would violate memory lifetime invariants due to
  // the ``destroy_addr %1`` above. But this would not violate the rules of
  // Ownership SSA since addresses exist outside of the guarantees of
  // Ownership SSA.
  //
  // %2 = load [take] %1 : $*Klass

  // I can return this object without worrying about needing to copy since
  // none objects can be arbitrarily returned.
  return %0 : $Int
}
```

Unowned

This is a form of ownership that is used to model two different use cases:

- Arguments of functions with ObjC convention. This convention requires the callee to copy the value before using it (preferably before any other code runs). We do not model this flow sensitive property in SIL today, but we do not allow for unowned values to be passed as owned or guaranteed values without copying it first.
- Values that are a conversion from a trivial value with None ownership to a non-trivial value. As an example of this consider an unsafe bit cast of a trivial pointer to a class. In that case, since we have no reason to assume that the object will remain alive, we need to make a copy of the value.

Ownership Constraint

NOTE: We assume that one has read the section above on Ownership Kind.

As mentioned above, every operand `operand(i)` of a SIL instruction `i` has statically mapped to it:

1. An ownership kind that acts as an "Ownership Constraint" upon what "Ownership Kind" a value can take.
2. A boolean value that defines whether or not the execution of the operand's instruction will cause the operand's value to be invalidated. This is often times referred to as an operand acting as a "lifetime ending use".

Forwarding Uses

NOTE: In the following, we assumed that one read the section above, Ownership Kind, Value Ownership Kind and Ownership Constraint.

A subset of SIL instructions define the value ownership kind of their results in terms of the value ownership kind of their operands. Such an instruction is called a "forwarding instruction" and any use with such a user instruction a "forwarding use". This inference generally occurs upon instruction construction and as a result:

- When manipulating forwarding instructions programmatically, one must manually update their forwarded ownership since most of the time the ownership is stored in the instruction itself. Don't worry though because the SIL verifier will catch this error for you if you forget to do so!
- Textual SIL does not represent the ownership of forwarding instructions explicitly. Instead, the instruction's ownership is inferred normally from the parsed operand. In some cases the forwarding ownership kind is different from the ownership kind of its operand. In such cases, textual SIL represents the forwarding ownership kind explicitly. Eg:

  ```
  %cast = unchecked_ref_cast %val : $Klass to $Optional<Klass>, forwarding: @unowned
  ```

  Since the SILVerifier runs on Textual SIL after parsing, you can feel confident that ownership constraints were inferred correctly.

Forwarding has slightly different ownership semantics depending on the value ownership kind of the operand on construction and the result's type. We go through each below:

- Given an `@owned` operand, the forwarding instruction is assumed to end the lifetime of the operand and produce an `@owned` value if non-trivially typed and `@none` if trivially typed. Example: This is used to represent the semantics of casts:

  ```
  sil @unsafelyCastToSubClass : $@convention(thin) (@owned Klass) -> @owned SubKlass {
  bb0(%0 : @owned $Klass): // %0 is defined here.

    // %0 is consumed here and can no longer be used after this point.
    // %1 is defined here and after this point must be used to access the object
    // passed in via %0.
    %1 = unchecked_ref_cast %0 : $Klass to $SubKlass

    // Then %1's lifetime ends here and we return the casted argument to our
    // caller as an @owned result.
    return %1 : $SubKlass
  }
  ```

- Given a `@guaranteed` operand, the forwarding instruction is assumed to produce `@guaranteed` non-trivially typed values and `@none` trivially typed values. Given the non-trivial case, the instruction is assumed to begin a new implicit borrow scope for the incoming value. Since the borrow scope is implicit, we validate the uses of the result as if they were uses of the operand (recursively). This of course means that one should never see end_borrows on any guaranteed forwarded results, the end_borrow is always on the instruction that "introduces" the borrowed value. An example of a guaranteed forwarding instruction is `struct_extract`:

  ```
  // In this function, I have a pair of Klasses and I want to grab some state
  // and then call the hand off function for someone else to continue
  // processing the pair.
  sil @accessLHSStateAndHandOff : $@convention(thin) (@owned KlassPair) -> @owned State {
  bb0(%0 : @owned $KlassPair): // %0 is defined here.

    // Begin the borrow scope for %0. We want to access %1's subfield in a
    // read only way that doesn't involve destructuring and extra copies. So
    // we construct a guaranteed scope here so we can safely use a
    // struct_extract.
    %1 = begin_borrow %0 : $KlassPair

    // Now we perform our struct_extract operation. This operation
    // structurally grabs a value out of a struct without safety relying on
    // the guaranteed ownership of its operand to know that %1 is live at all
    // use points of %2, its result.
    %2 = struct_extract %1 : $KlassPair, #KlassPair.lhs

    // Then grab the state from our left hand side klass and copy it so we
    // can pass off our klass pair to handOff for continued processing.
    %3 = ref_element_addr %2 : $Klass, #Klass.state
    %4 = load [copy] %3 : $*State

    // Now that we have finished accessing %1, we end the borrow scope for %1.
    end_borrow %1 : $KlassPair

    %handOff = function_ref @handOff : $@convention(thin) (@owned KlassPair) -> ()
    apply %handOff(%0) : $@convention(thin) (@owned KlassPair) -> ()

    return %4 : $State
  }
  ```

- Given an `@none` operand, the result value must have `@none` ownership.
- Given an `@unowned` operand, the result value will have `@unowned` ownership. It will be validated just like any other `@unowned` value, namely that it must be copied before use.

An additional wrinkle here is that even though the vast majority of forwarding instructions forward all types of ownership, this is not true in general. To see why this is necessary, lets compare/contrast struct_extract (which does not forward `@owned` ownership) and unchecked_enum_data (which can forward /all/ ownership kinds). The reason for this difference is that struct_extract inherently can only extract out a single field of a larger object implying that the instruction could only represent consuming a sub-field of a value

instead of the entire value at once. This violates our constraint that owned values can never be partially consumed: a value is either completely alive or completely dead. In contrast, enums always represent their payloads as elements in a single tuple value. This means that unchecked_enum_data when it extracts that payload from an enum, can consume the entire enum+payload.

To handle cases where we want to use struct_extract in a consuming way, we instead are able to use the destructure_struct instruction that consumes the entire struct at once and gives one back the structs individual constituant parts:

```
struct KlassPair {
  var fieldOne: Klass
  var fieldTwo: Klass
}

sil @getFirstPairElt : $@convention(thin) (@owned KlassPair) -> @owned Klass {
bb0(%0 : @owned $KlassPair):
  // If we were to just do this directly and consume KlassPair to access
  // fieldOne... what would happen to fieldTwo? Would it be consumed?
  //
  // %1 = struct_extract %0 : $KlassPair, #KlassPair.fieldOne
  //
  // Instead we need to destructure to ensure we consume the entire owned value at once.
  (%1, %2) = destructure_struct $KlassPair

  // We only want to return %1, so we need to cleanup %2.
  destroy_value %2 : $Klass

  // Then return %1 to our caller
  return %1 : $Klass
}
```

Forwarding Address-Only Values

Address-only values are potentially unmovable when borrowed. This means that they cannot be forwarded with guaranteed ownership unless the forwarded value has the same representation as in the original value and can reuse the same storage. Non-destructive projection is allowed, such as *struct_extract*. Aggregation, such as *struct*, and destructive disaggregation, such as *switch_enum* is not allowed. This is an invariant for OSSA with opaque SIL values for these reasons:

1. To avoid implicit semantic copies. For move-only values, this allows complete diagnostics. And in general, it makes it impossible for SIL passes to "accidentally" create copies.

2. To reuse borrowed storage. This allows the optimizer to share the same storage for multiple exclusive reads of the same variable, avoiding copies. It may also be necessary to support native Swift atomics, which will be unmovable-when-borrowed.

Borrowed Object based Safe Interior Pointers

What is an "Unsafe Interior Pointer"

An unsafe interior pointer is a bare pointer into the innards of an object. A simple example of this in C++ would be using the method std::vector::data() to get to the innards of a std::vector. In general interior pointers are unsafe to use since languages do not provide any guarantees that the interior pointer will not be used after the underlying object has been deallocated. To see this, consider the following C++ example:

```
int unfortunateFunction() {
  int *unsafeInteriorPointer = nullptr;
  {
    std::vector<int> vector;
    vector.push_back(5);
    unsafeInteriorPointer = vector.data();
    printf("%d\n", *unsafeInteriorPointer); // Prints "5".
  } // vector deallocated here
  return *unsafeInteriorPointer; // Kaboom
}
```

In words, C++ allows for us to get the interior pointer into the vector, but then lets us do whatever we want with the pointer, including use it after the underlying memory has been invalidated.

From a user's perspective, interior pointers are really useful since one can use it to pass data to other APIs that are only expecting a pointer and also since one can use it to sometimes get better performance. But from a language designer perspective, this sort of API verboten and leads to bugs, crashes, and security vulnerabilities. That being said, clearly users have a need for such functionality, so we, as language designers, should figure out manners to express these sorts of patterns in our various languages in a safe way that prevents userâ€™s from foot-gunning themselves. In SIL, we have solved this problem via the direct modeling of interior pointer instructions as a high level concept in our IR.

Safe Interior Pointers in SIL

In contrast to LLVM-IR, SIL provides mechanisms that language designers can use to express concepts like the above in a manner that allows the language to define away compiler generated unsafe interior pointer usage using "Safe Interior Pointers". This is implemented in SIL by:

1. Classifying a set of instructions as being "interior pointer" instructions.
2. Enforcing in the SILVerifier that all "interior pointer" instructions can only have operands with Guaranteed ownership.
3. Enforcing in the SILVerifier that any transitive address use of the interior pointer to be a liveness requirement of the "interior pointer"s operand.

Note that the transitive address use verifier from (3) does not attempt to classify uses directly. Instead the verifier:

1. Has an explicit list of instructions that it understands as requiring liveness of the base object.
2. Has a second list of instructions that require liveness and produce a address whose transitive uses need to be recursively processed.
3. Asserts on any instructions that are not known to the verifier. This ensures that the verifier is kept up to date with new instructions.

Note that typically instructions in category (1) are instructions whose uses do not propagate the pointer value, so they are safe. In contrast, some other instructions in category (1) are escaping uses of the address such as pointer_to_address. Those uses are unsafe--the user is reponsible for managing unsafe pointer lifetimes and the compiler must not extend those pointer lifetimes.

These rules ensure statically that any uses of the address that are not escaped explicitly by an instruction like pointer_to_address are within the guaranteed pointers scope where the guaranteed value is statically known to be live. As a result, in SIL it is impossible to express such a bug in compiler generated code. As an example, consider the following unsafe interior pointer SIL:

```
class Klass { var k: KlassField }
struct KlassWrapper { var k: Klass }

// ...

// Today SIL restricts interior pointer instructions to only have operands
// with guaranteed ownership.
%1 = begin_borrow %0 : $Klass

// %2 is an interior pointer into %1. Since %2 is an address, it's uses are
// not treated as uses of underlying borrowed object %1 in the ownership
// system. This is because at the ownership level objects with None
// ownership are not verified and do not have any constraints on how they
// are used from the ownership system.
//
// Instead the ownership verifier gathers up all such uses and treats them
// as uses of the object from which the interior pointer was projected from
// transitively. This means that this is a constraint on the guaranteed
// objects use, not on the trivial values.
%2 = ref_element_addr %1 : $Klass, #Klass.k // %2 is a $*KlassWrapper
%3 = struct_element_addr %2 : $*KlassWrapper, #KlassWrapper.k // %3 is a $*Klass

// So if we end the borrow %1 at this point, invalidating the addresses
// ``%2`` and ``%3``.
end_borrow %1 : $Klass
```

```
// We would here be loading from an invalidated address. This would cause a
// verifier error since %3's use here is a regular use that is inferred up
// on %1.
%4 = load [copy] %3 : $*KlassWrapper

// ...
```

Notice how due to a possible bug in the compiler, we are loading from potentially uninitialized memory `%4`. This would have caused a verifier error stating that `%4` was an interior pointer based use-after-free of `%1` implying this is mal-formed SIL.

NOTE: This is a constraint on the base object, not on the addresses themselves which are viewed as outside of the ownership system since they have None ownership.

In contrast to the previous example, the following example follows ownership invariants and is valid SIL:

```
class Klass { var k: KlassField }
struct KlassWrapper { var k: Klass }

// ...

%1 = begin_borrow %0 : $Klass
// %2 is an interior pointer into the Klass k. Since %2 is an address and
// addresses have None ownership, it's uses are not treated as uses of the
// underlying object %1.
%2 = ref_element_addr %1 : $Klass, #Klass.k // %2 is a $*KlassWrapper

// Destroying %1 at this location would result in a verifier error since
// %2's uses are considered to be uses of %1.
//
// end_lifetime %1 : $Klass

// We are statically not loading from an invalidated address here since we
// are within the lifetime of ``%1``.
%3 = struct_element_addr %2 : $*KlassWrapper, #KlassWrapper.k
%4 = load [copy] %3 : $*Klass // %1 must be live here transitively

// ``%1``'s lifetime ends. Importantly we know that within the lifetime of
// ``%1``, ``%0``'s lifetime can not shrink past this point, implying
// transitive static safety.
end_borrow %1 : $Klass
```

In the second example, we show a well-formed SIL program showing off SIL's Safe Interior Pointers. All of the uses of `%2`, the interior pointer, are transitively uses of the base underlying object, `%0`.

The current list of interior pointer SIL instructions are:

- project_box - projects a pointer out of a reference counted box. (*)
- ref_element_addr - projects a field out of a reference counted class.
- ref_tail_addr - projects out a pointer to a classâ€™s tail allocated array memory (assuming the class was initialized to have such an array).
- open_existential_box - projects the address of the value out of a boxed existential container using the current function context/protocol conformance to create an "opened archetype".
- project_existential_box - projects a pointer to the value inside a boxed existential container. Must be the type for which the box was initially allocated for and not for an "opened" archetype.

(*) We still need to finish adding support for project_box, but all other interior pointers are guarded already.

Variable Lifetimes

In order for programmer intended lifetimes to be maintained under optimization, the lifetimes of SIL values which correspond to named source-level values can only be modified in limited ways. Generally, the behavior is that the lifetime of a named source-level value cannot _observably_ end before the end of the lexical scope in which that value is defined. Specifically, code motion may not move the ends of these lifetimes across a **deinit barrier**.

A few sorts of SIL value have lifetimes that are constrained that way:

1: *begin_borrow [lexical]* 2: *move_value [lexical]* 3: @owned function arguments 4: *alloc_stack [lexical]*

That these three have constrained lifetimes is encoded in ValueBase::isLexical, which should be checked before changing the lifetime of a value.

The reason that only @owned function arguments are constrained is that a @guaranteed function argument is guaranteed by the function's caller to live for the full duration of the function already. Optimization of the function alone can't shorten it. When such a function is inlined into its caller, though, a lexical borrow scope is added for each of its @guaranteed arguments, ensuring that the lifetime of the corresponding source-level value is not shortened in a way that doesn't respect deinit barriers.

Unlike the other sorts, *alloc_stack [lexical]* isn't a SILValue. Instead, it constrains the lifetime of an addressable variable. Since the constraint is applied to the in-memory representation, no additional lexical SILValue is required.

Deinit Barriers

Deinit barriers (see swift::isDeinitBarrier) are instructions which would be affected by the side effects of deinitializers. To maintain the order of effects that is visible to the programmer, destroys of lexical values cannot be reordered with respect to them. There are three kinds:

1. synchronization points (locks, memory barriers, syscalls, etc.)
2. loads of weak or unowned values
3. accesses of pointers

Examples:

1. Given an instance of a class which owns a file handle and closes the file handle on deinit, writing to the file handle and then deallocating the instance would result in changes being written. If the destroy of the instance were hoisted above the call to write to the file handle, an error would be raised instead.
2. Given an instance *c* of a class *C* which weakly references an instance *d* of a second class *D*, if *d* is referenced via a local variable *v*, then loading that weak reference from *c* within the variable scope should return a non-nil reference to *d*. Hoisting the destroy of *v* above the weak load from *c*, however, would result in the destruction of *d* before that load and a nil weak reference to *D*.
3. Given an instance of a class which owns a buffer and deallocates it on deinitialization, accessing the pointer and then deallocating the instance is defined behavior. Hoisting the destroy of the instance above the access to the memory would result in accessing a freed pointer.

Memory Lifetime

Similar to Ownership SSA, there are also lifetime rules for values in memory. With "memory" we refer to memory which is addressed by SIL instruction with address-type operands, like `load`, `store`, `switch_enum_addr`, etc.

Each memory location which holds a non-trivial value is either uninitialized or initialized. A memory location gets initialized by storing values into it (except assignment, which expects a location to be already initialized). A memory location gets de-initialized by "taking" from it or destroying it, e.g. with `destroy_addr`. It is illegal to re-initialize a memory location or to use a location after it was de-initialized.

If a memory location holds a trivial value (e.g. an `Int`), it is not required to de-initialize the location.

The SIL verifier checks this rule for memory locations which can be uniquely identified, for example and `alloc_stack` or an indirect parameter. The verifier cannot check memory locations which are potentially aliased, e.g. a `ref_element_addr` (a stored class property).

Lifetime of Enums in Memory

The situation is a bit more complicated with enums, because an enum can have both, cases with non-trivial payloads and cases with no payload or trivial payloads.

Even if an enum itself is not trivial (because it has at least on case with a non-trivial payload), it is not required to de-initialize such an enum memory location on paths where it's statically provable that the enum contains a trivial or non-payload case.

That's the case if the destroy point is jointly dominated by:

- a `store [trivial]` to the enum memory location.

or

- an `inject_enum_addr` to the enum memory location with a non-trivial or non-payload case.

or

- a successor of a `switch_enum` or `switch_enum_addr` for a non-trivial or non-payload case.

Dead End Blocks

In SIL, one can express that a program is semantically expected to exit at the end of a block by terminating the block with an unreachable. Such a block is called a *program terminating block* and all blocks that are post-dominated by blocks of the aforementioned kind are called *dead end blocks*. Intuitively, any path through a dead end block is known to result in program termination, so resources that normally would need to be released back to the system will instead be returned to the system by process tear down.

Since we rely on the system at these points to perform resource cleanup, we are able to loosen our lifetime requirements by allowing for values to not have their lifetimes ended along paths that end in program terminating blocks. Operationally, this implies that:

- All SIL values must have exactly one lifetime ending use on all paths that terminate in a return or throw. In contrast, a SIL value does not need to have a lifetime ending use along paths that end in an unreachable.
- end_borrow and destroy_value are redundent, albeit legal, in blocks where all paths through the block end in an unreachable.

Consider the following legal SIL where we leak `%0` in blocks prefixed with `bbDeadEndBlock` and consume it in `bb2`:

```
sil @user : $@convention(thin) (@owned Klass) -> @owned Klass {
bb0(%0 : @owned $Klass):
  cond_br ..., bb1, bb2

bb1:
  // This is a dead end block since it is post-dominated by two dead end
  // blocks. It is not a program terminating block though since the program
  // does not end in this block.
  cond_br ..., bbDeadEndBlock1, bbDeadEndBlock2

bbDeadEndBlock1:
  // This is a dead end block and a program terminating block.
  //
  // We are exiting the program here causing the operating system to clean up
  // all resources associated with our process, so there is no need for a
  // destroy_value. That memory will be cleaned up anyways.
  unreachable

bbDeadEndBlock2:
  // This is a dead end block and a program terminating block.
  //
  // Even though we do not need to insert destroy_value along these paths, we
  // can if we want to. It is just necessary and the optimizer can eliminate
  // such a destroy_value if it wishes.
  //
  // NOTE: The author arbitrarily chose just to destroy %0: we could legally
  // destroy either value (or both!).
  destroy_value %0 : $Klass
  unreachable

bb2:
  cond_br ..., bb3, bb4

bb3:
  // This block is live, so we need to ensure that %0 is consumed within the
  // block. In this case, %0 is consumed by returning %0 to our caller.
  return %0 : $Klass

bb4:
  // This block is also live, but since we do not return %0, we must insert a
  // destroy_value to cleanup %0.
  //
  // NOTE: The copy_value/destroy_value here is redundent and can be removed by
  // the optimizer. The author left it in for illustrative purposes.
  %1 = copy_value %0 : $Klass
  destroy_value %0 : $Klass
  return %1 : $Klass
}
```

Runtime Failure

Some operations, such as failed unconditional checked conversions or the `Builtin.trap` compiler builtin, cause a *runtime failure*, which unconditionally terminates the current actor. If it can be proven that a runtime failure will occur or did occur, runtime failures may be reordered so long as they remain well-ordered relative to operations external to the actor or the program as a whole. For instance, with overflow checking on integer arithmetic enabled, a simple `for` loop that reads inputs in from one or more arrays and writes outputs to another array, all local to the current actor, may cause runtime failure in the update operations:

```
// Given unknown start and end values, this loop may overflow
for var i = unknownStartValue; i != unknownEndValue; ++i {
  ...
}
```

It is permitted to hoist the overflow check and associated runtime failure out of the loop itself and check the bounds of the loop prior to entering it, so long as the loop body has no observable effect outside of the current actor.

Undefined Behavior

Incorrect use of some operations is *undefined behavior*, such as invalid unchecked casts involving `Builtin.RawPointer` types, or use of compiler builtins that lower to LLVM instructions with undefined behavior at the LLVM level. A SIL program with undefined behavior is meaningless, much like undefined behavior in C, and has no predictable semantics. Undefined behavior should not be triggered by valid SIL emitted by a correct Swift program using a correct standard library, but cannot in all cases be diagnosed or verified at the SIL level.

Calling Convention

This section describes how Swift functions are emitted in SIL.

Swift Calling Convention @convention(swift)

The Swift calling convention is the one used by default for native Swift functions.

Tuples in the input type of the function are recursively destructured into separate arguments, both in the entry point basic block of the callee, and in the `apply` instructions used by callers:

```
func foo(_ x:Int, y:Int)

sil @foo : $(x:Int, y:Int) -> () {
entry(%x : $Int, %y : $Int):
  ...
}

func bar(_ x:Int, y:(Int, Int))
```

```
sil @bar : $(x:Int, y:(Int, Int)) -> () {
entry(%x : $Int, %y0 : $Int, %y1 : $Int):
  ...
}

func call_foo_and_bar() {
  foo(1, 2)
  bar(4, (5, 6))
}

sil @call_foo_and_bar : $() -> () {
entry:
  ...
  %foo = function_ref @foo : $(x:Int, y:Int) -> ()
  %foo_result = apply %foo(%1, %2) : $(x:Int, y:Int) -> ()
  ...
  %bar = function_ref @bar : $(x:Int, y:(Int, Int)) -> ()
  %bar_result = apply %bar(%4, %5, %6) : $(x:Int, y:(Int, Int)) -> ()
}
```

Calling a function with trivial value types as inputs and outputs simply passes the arguments by value. This Swift function:

```
func foo(_ x:Int, y:Float) -> UnicodeScalar

foo(x, y)
```

gets called in SIL as:

```
%foo = constant_ref $(Int, Float) -> UnicodeScalar, @foo
%z = apply %foo(%x, %y) : $(Int, Float) -> UnicodeScalar
```

Reference Counts

NOTE This section only is speaking in terms of rules of thumb. The actual behavior of arguments with respect to arguments is defined by the argument's convention attribute (e.g. `@owned`), not the calling convention itself.

Reference type arguments are passed in at +1 retain count and consumed by the callee. A reference type return value is returned at +1 and consumed by the caller. Value types with reference type components have their reference type components each retained and released the same way. This Swift function:

```
class A {}

func bar(_ x:A) -> (Int, A) { ... }

bar(x)
```

gets called in SIL as:

```
%bar = function_ref @bar : $(A) -> (Int, A)
strong_retain %x : $A
%z = apply %bar(%x) : $(A) -> (Int, A)
// ... use %z ...
%z_1 = tuple_extract %z : $(Int, A), 1
strong_release %z_1
```

When applying a thick function value as a callee, the function value is also consumed at +1 retain count.

Address-Only Types

For address-only arguments, the caller allocates a copy and passes the address of the copy to the callee. The callee takes ownership of the copy and is responsible for destroying or consuming the value, though the caller must still deallocate the memory. For address-only return values, the caller allocates an uninitialized buffer and passes its address as the first argument to the callee. The callee must initialize this buffer before returning. This Swift function:

```
 @API struct A {}

func bas(_ x:A, y:Int) -> A { return x }

var z = bas(x, y)
// ... use z ...
```

gets called in SIL as:

```
%bas = function_ref @bas : $(A, Int) -> A
%z = alloc_stack $A
%x_arg = alloc_stack $A
copy_addr %x to [initialize] %x_arg : $*A
apply %bas(%z, %x_arg, %y) : $(A, Int) -> A
dealloc_stack %x_arg : $*A // callee consumes %x.arg, caller deallocs
// ... use %z ...
destroy_addr %z : $*A
dealloc_stack stack %z : $*A
```

The implementation of `@bas` is then responsible for consuming `%x_arg` and initializing `%z`.

Tuple arguments are destructured regardless of the address-only-ness of the tuple type. The destructured fields are passed individually according to the above convention. This Swift function:

```
@API struct A {}

func zim(_ x:Int, y:A, (z:Int, w:(A, Int)))

zim(x, y, (z, w))
```

gets called in SIL as:

```
%zim = function_ref @zim : $(x:Int, y:A, (z:Int, w:(A, Int))) -> ()
%y_arg = alloc_stack $A
copy_addr %y to [initialize] %y_arg : $*A
%w_0_addr = element_addr %w : $*(A, Int), 0
%w_0_arg = alloc_stack $A
copy_addr %w_0_addr to [initialize] %w_0_arg : $*A
%w_1_addr = element_addr %w : $*(A, Int), 1
%w_1 = load %w_1_addr : $*Int
apply %zim(%x, %y_arg, %z, %w_0_arg, %w_1) : $(x:Int, y:A, (z:Int, w:(A, Int))) -> ()
dealloc_stack %w_0_arg
dealloc_stack %y_arg
```

Variadic Arguments

Variadic arguments and tuple elements are packaged into an array and passed as a single array argument. This Swift function:

```
func zang(_ x:Int, (y:Int, z:Int...), v:Int, w:Int...)

zang(x, (y, z0, z1), v, w0, w1, w2)
```

gets called in SIL as:

```
%zang = function_ref @zang : $(x:Int, (y:Int, z:Int...), v:Int, w:Int...) -> ()
%zs = <<make array from %z1, %z2>>
%ws = <<make array from %w0, %w1, %w2>>
apply %zang(%x, %y, %zs, %v, %ws)  : $(x:Int, (y:Int, z:Int...), v:Int, w:Int...) -> ()
```

@inout Arguments

`@inout` arguments are passed into the entry point by address. The callee does not take ownership of the referenced memory. The referenced memory must be initialized upon function entry and exit. If the `@inout` argument refers to a fragile physical variable, then the argument is the address of that variable. If the `@inout` argument refers to a logical property, then the argument is the address of a

caller-owned writeback buffer. It is the caller's responsibility to initialize the buffer by storing the result of the property getter prior to calling the function and to write back to the property on return by loading from the buffer and invoking the setter with the final value. This Swift function:

```
func inout(_ x: inout Int) {
  x = 1
}
```

gets lowered to SIL as:

```
sil @inout : $(@inout Int) -> () {
entry(%x : $*Int):
  %1 = integer_literal $Int, 1
  store %1 to %x
  return
}
```

Swift Method Calling Convention @convention(method)

The method calling convention is currently identical to the freestanding function convention. Methods are considered to be curried functions, taking the "self" argument as their outer argument clause, and the method arguments as the inner argument clause(s). The "self" argument is thus passed last:

```
struct Foo {
  func method(_ x:Int) -> Int {}
}

sil @Foo_method_1 : $((x : Int), @inout Foo) -> Int { ... }
```

Witness Method Calling Convention @convention(witness_method)

The witness method calling convention is used by protocol witness methods in witness tables. It is identical to the `method` calling convention except that its handling of generic type parameters. For non-witness methods, the machine-level convention for passing type parameter metadata may be arbitrarily dependent on static aspects of the function signature, but because witnesses must be polymorphically dispatchable on their `Self` type, the `Self`-related metadata for a witness must be passed in a maximally abstracted manner.

C Calling Convention @convention(c)

In Swift's C module importer, C types are always mapped to Swift types considered trivial by SIL. SIL does not concern itself with platform ABI requirements for indirect return, register vs. stack passing, etc.; C function arguments and returns in SIL are always by value regardless of the platform calling convention.

SIL (and therefore Swift) cannot currently invoke variadic C functions.

Objective-C Calling Convention @convention(objc_method)

Reference Counts

Objective-C methods use the same argument and return value ownership rules as ARC Objective-C. Selector families and the `ns_consumed`, `ns_returns_retained`, etc. attributes from imported Objective-C definitions are honored.

Applying a `@convention(block)` value does not consume the block.

Method Currying

In SIL, the "self" argument of an Objective-C method is uncurried to the last argument of the uncurried type, just like a native Swift method:

```
@objc class NSString {
  func stringByPaddingToLength(Int) withString(NSString) startingAtIndex(Int)
}

sil @NSString_stringByPaddingToLength_withString_startingAtIndex \
  : $((Int, NSString, Int), NSString)
```

That `self` is passed as the first argument at the IR level is abstracted away in SIL, as is the existence of the `_cmd` selector argument.

Type Based Alias Analysis

SIL supports two types of Type Based Alias Analysis (TBAA): Class TBAA and Typed Access TBAA.

Class TBAA

Class instances and other *heap object references* are pointers at the implementation level, but unlike SIL addresses, they are first class values and can be `capture`-d and aliased. Swift, however, is memory-safe and statically typed, so aliasing of classes is constrained by the type system as follows:

- A `Builtin.NativeObject` may alias any native Swift heap object, including a Swift class instance, a box allocated by `alloc_box`, or a thick function's closure context. It may not alias natively Objective-C class instances.
- An `AnyObject` or `Builtin.BridgeObject` may alias any class instance, whether Swift or Objective-C, but may not alias non-class-instance heap objects.
- Two values of the same class type `$C` may alias. Two values of related class type `$B` and `$D`, where there is a subclass relationship between `$B` and `$D`, may alias. Two values of unrelated class types may not alias. This includes different instantiations of a generic class type, such as `$C<Int>` and `$C<Float>`, which currently may never alias.
- Without whole-program visibility, values of archetype or protocol type must be assumed to potentially alias any class instance. Even if it is locally apparent that a class does not conform to that protocol, another component may introduce a conformance by an extension. Similarly, a generic class instance, such as `$C<T>` for archetype `T`, must be assumed to potentially alias concrete instances of the generic type, such as `$C<Int>`, because `Int` is a potential substitution for `T`.

A violation of the above aliasing rules only results in undefined behavior if the aliasing references are dereferenced within Swift code. For example, `__SwiftNativeNS[Array|Dictionary|String]` classes alias with `NS[Array|Dictionary|String]` classes even though they are not statically related. Since Swift never directly accesses stored properties on the Foundation classes, this aliasing does not pose a danger.

Typed Access TBAA

Define a *typed access* of an address or reference as one of the following:

- Any instruction that performs a typed read or write operation upon the memory at the given location (e.x. `load`, `store`).
- Any instruction that yields a typed offset of the pointer by performing a typed projection operation (e.x. `ref_element_addr`, `tuple_element_addr`).

With limited exceptions, it is undefined behavior to perform a typed access to an address or reference addressed memory is not bound to the relevant type.

This allows the optimizer to assume that two addresses cannot alias if there does not exist a substitution of archetypes that could cause one of the types to be the type of a subobject of the other. Additionally, this applies to the types of the values from which the addresses were derived via a typed projection.

Consider the following SIL:

```
struct Element {
  var i: Int
}
struct S1 {
  var elt: Element
}
struct S2 {
  var elt: Element
```

```
}
%adr1 = struct_element_addr %ptr1 : $*S1, #S.elt
%adr2 = struct_element_addr %ptr2 : $*S2, #S.elt
```

The optimizer may assume that `%adr1` does not alias with `%adr2` because the values that the addresses are derived from (`%ptr1` and `%ptr2`) have unrelated types. However, in the following example, the optimizer cannot assume that `%adr1` does not alias with `%adr2` because `%adr2` is derived from a cast, and any subsequent typed operations on the address will refer to the common `Element` type:

```
%adr1 = struct_element_addr %ptr1 : $*S1, #S.elt
%adr2 = pointer_to_address %ptr2 : $Builtin.RawPointer to $*Element
```

Exceptions to typed access TBAA rules are only allowed for blessed alias-introducing operations. This permits limited type-punning. The only current exception is the non-struct `pointer_to_address` variant. The optimizer must be able to defensively determine that none of the *roots* of an address are alias-introducing operations. An address root is the operation that produces the address prior to applying any typed projections, indexing, or casts. The following are valid address roots:

- Object allocation that generates an address, such as `alloc_stack` and `alloc_box`.
- Address-type function arguments. These are crucially *not* considered alias-introducing operations. It is illegal for the SIL optimizer to form a new function argument from an arbitrary address-type value. Doing so would require the optimizer to guarantee that the new argument is both has a non-alias-introducing address root and can be properly represented by the calling convention (address types do not have a fixed representation).
- A strict cast from an untyped pointer, `pointer_to_address [strict]`. It is illegal for `pointer_to_address [strict]` to derive its address from an alias-introducing operation's value. A type punned address may only be produced from an opaque pointer via a non-strict `pointer_to_address` at the point of conversion.

Address-to-address casts, via `unchecked_addr_cast`, transparently forward their source's address root, just like typed projections.

Address-type basic block arguments can be conservatively considered aliasing-introducing operations; they are uncommon enough not to matter and may eventually be prohibited altogether.

Although some pointer producing intrinsics exist, they do not need to be considered alias-introducing exceptions to TBAA rules. `Builtin.inttoptr` produces a `Builtin.RawPointer` which is not interesting because by definition it may alias with everything. Similarly, the LLVM builtins `Builtin.bitcast` and `Builtin.trunc|sext|zextBitCast` cannot produce typed pointers. These pointer values must be converted to an address via `pointer_to_address` before typed access can occur. Whether the `pointer_to_address` is strict determines whether aliasing may occur.

Memory may be rebound to an unrelated type. Addresses to unrelated types may alias as long as typed access only occurs while memory is bound to the relevant type. Consequently, the optimizer cannot outright assume that addresses accessed as unrelated types are nonaliasing. For example, pointer comparison cannot be eliminated simply because the two addresses derived from those pointers are accessed as unrelated types at different program points.

Value Dependence

In general, analyses can assume that independent values are independently assured of validity. For example, a class method may return a class reference:

```
bb0(%0 : $MyClass):
  %1 = class_method %0 : $MyClass, #MyClass.foo
  %2 = apply %1(%0) : $@convention(method) (@guaranteed MyClass) -> @owned MyOtherClass
  // use of %2 goes here; no use of %1
  strong_release %2 : $MyOtherClass
  strong_release %1 : $MyClass
```

The optimizer is free to move the release of `%1` to immediately after the call here, because `%2` can be assumed to be an independently-managed value, and because Swift generally permits the reordering of destructors.

However, some instructions do create values that are intrinsically dependent on their operands. For example, the result of `ref_element_addr` will become a dangling pointer if the base is released too soon. This is captured by the concept of *value dependence*, and any transformation which can reorder of destruction of a value around another operation must remain conscious of it.

A value `%1` is said to be *value-dependent* on a value `%0` if:

- `%1` is the result and `%0` is the first operand of one of the following instructions:
  - `ref_element_addr`
  - `struct_element_addr`
  - `tuple_element_addr`
  - `unchecked_take_enum_data_addr`
  - `pointer_to_address`
  - `address_to_pointer`
  - `index_addr`
  - `index_raw_pointer`
  - possibly some other conversions
- `%1` is the result of `mark_dependence` and `%0` is either of the operands.
- `%1` is the value address of a box allocation instruction of which `%0` is the box reference.
- `%1` is the result of a `struct`, `tuple`, or `enum` instruction and `%0` is an operand.
- `%1` is the result of projecting out a subobject of `%0` with `tuple_extract`, `struct_extract`, `unchecked_enum_data`, `select_enum`, or `select_enum_addr`.
- `%1` is the result of `select_value` and `%0` is one of the cases.
- `%1` is a basic block parameter and `%0` is the corresponding argument from a branch to that block.
- `%1` is the result of a `load` from `%0`. However, the value dependence is cut after the first attempt to manage the value of `%1`, e.g. by retaining it.
- Transitivity: there exists a value `%2` which `%1` depends on and which depends on `%0`. However, transitivity does not apply to different subobjects of a struct, tuple, or enum.

Note, however, that an analysis is not required to track dependence through memory. Nor is it required to consider the possibility of dependence being established "behind the scenes" by opaque code, such as by a method returning an unsafe pointer to a class property. The dependence is required to be locally obvious in a function's SIL instructions. Precautions must be taken against this either by SIL generators (by using `mark_dependence` appropriately) or by the user (by using the appropriate intrinsics and attributes with unsafe language or library features).

Only certain types of SIL value can carry value-dependence:

- SIL address types
- unmanaged pointer types:
  - `@sil_unmanaged` types
  - `Builtin.RawPointer`
  - aggregates containing such a type, such as `UnsafePointer`, possibly recursively
- non-trivial types (but they can be independently managed)

This rule means that casting a pointer to an integer type breaks value-dependence. This restriction is necessary so that reading an `Int` from a class doesn't force the class to be kept around! A class holding an unsafe reference to an object must use some sort of unmanaged pointer type to do so.

This rule does not include generic or resilient value types which might contain unmanaged pointer types. Analyses are free to assume that e.g. a `copy_addr` of a generic or resilient value type yields an independently-managed value. The extension of value dependence to types containing obvious unmanaged pointer types is an affordance to make the use of such types more convenient; it does not shift the ultimate responsibility for assuring the safety of unsafe language/library features away from the user.

Copy-on-Write Representation

Copy-on-Write (COW) data structures are implemented by a reference to an object which is copied on mutation in case it's not uniquely referenced.

A COW mutation sequence in SIL typically looks like:

```
  (%uniq, %buffer) = begin_cow_mutation %immutable_buffer : $BufferClass
  cond_br %uniq, bb_uniq, bb_not_unique
bb_uniq:
  br bb_mutate(%buffer : $BufferClass)
bb_not_unique:
  %copied_buffer = apply %copy_buffer_function(%buffer) : ...
  br bb_mutate(%copied_buffer : $BufferClass)
bb_mutate(%mutable_buffer : $BufferClass):
  %field = ref_element_addr %mutable_buffer : $BufferClass, #BufferClass.Field
  store %value to %field : $ValueType
  %new_immutable_buffer = end_cow_mutation %buffer : $BufferClass
```

Loading from a COW data structure looks like:

```
%field1 = ref_element_addr [immutable] %immutable_buffer : $BufferClass, #BufferClass.Field
%value1 = load %field1 : $*FieldType
...
%field2 = ref_element_addr [immutable] %immutable_buffer : $BufferClass, #BufferClass.Field
%value2 = load %field2 : $*FieldType
```

The `immutable` attribute means that loading values from `ref_element_addr` and `ref_tail_addr` instructions, which have the *same* operand, are equivalent. In other words, it's guaranteed that a buffer's properties are not mutated between two `ref_element/tail_addr [immutable]` as long as they have the same buffer reference as operand. This is even true if e.g. the buffer 'escapes' to an unknown function.

In the example above, `%value2` is equal to `%value1` because the operand of both `ref_element_addr` instructions is the same `%immutable_buffer`. Conceptually, the content of a COW buffer object can be seen as part of the same *static* (immutable) SSA value as the buffer reference.

The lifetime of a COW value is strictly separated into *mutable* and *immutable* regions by `begin_cow_mutation` and `end_cow_mutation` instructions:

```
%b1 = alloc_ref $BufferClass
// The buffer %b1 is mutable
%b2 = end_cow_mutation %b1 : $BufferClass
// The buffer %b2 is immutable
(%u1, %b3) = begin_cow_mutation %b1 : $BufferClass
// The buffer %b3 is mutable
%b4 = end_cow_mutation %b3 : $BufferClass
// The buffer %b4 is immutable
...
```

Both, `begin_cow_mutation` and `end_cow_mutation`, consume their operand and return the new buffer as an *owned* value. The `begin_cow_mutation` will compile down to a uniqueness check and `end_cow_mutation` will compile to a no-op.

Although the physical pointer value of the returned buffer reference is the same as the operand, it's important to generate a *new* buffer reference in SIL. It prevents the optimizer from moving buffer accesses from a *mutable* into a *immutable* region and vice versa.

Because the buffer *content* is conceptually part of the buffer *reference* SSA value, there must be a new buffer reference every time the buffer content is mutated.

To illustrate this, let's look at an example, where a COW value is mutated in a loop. As with a scalar SSA value, also mutating a COW buffer will enforce a phi-argument in the loop header block (for simplicity the code for copying a non-unique buffer is not shown):

```
header_block(%b_phi : $BufferClass):
  (%u, %b_mutate) = begin_cow_mutation %b_phi : $BufferClass
  // Store something to %b_mutate
  %b_immutable = end_cow_mutation %b_mutate : $BufferClass
  cond_br %loop_cond, exit_block, backedge_block
backedge_block:
  br header_block(b_immutable : $BufferClass)
exit_block:
```

Two adjacent `begin_cow_mutation` and `end_cow_mutation` instructions don't need to be in the same function.

Instruction Set

Allocation and Deallocation

These instructions allocate and deallocate memory.

alloc_stack

```
sil-instruction ::= 'alloc_stack' '[dynamic_lifetime]'? '[lexical]'? '[moved]'? sil-type (',' debug-var-attr)*

%1 = alloc_stack $T
// %1 has type $*T
```

Allocates uninitialized memory that is sufficiently aligned on the stack to contain a value of type `T`. The result of the instruction is the address of the allocated memory.

`alloc_stack` always allocates memory on the stack even for runtime-sized type.

`alloc_stack` marks the start of the lifetime of the value; the allocation must be balanced with a `dealloc_stack` instruction to mark the end of its lifetime. All `alloc_stack` allocations must be deallocated prior to returning from a function. If a block has multiple predecessors, the stack height and order of allocations must be consistent coming from all predecessor blocks. `alloc_stack` allocations must be deallocated in last-in, first-out stack order.

The `dynamic_lifetime` attribute specifies that the initialization and destruction of the stored value cannot be verified at compile time. This is the case, e.g. for conditionally initialized objects.

The optional `lexical` attribute specifies that the storage corresponds to a local variable in the Swift source.

The optional `moved` attribute specifies that at the source level, the variable associated with this alloc_stack was moved and furthermore that at the SIL level it passed move operator checking. This means that one can not assume that the value in the alloc_stack can be semantically valid over the entire function frame when emitting debug info.

The memory is not retainable. To allocate a retainable box for a value type, use `alloc_box`.

alloc_ref

```
sil-instruction ::= 'alloc_ref'
                    ('[' 'objc' ']')?
                    ('[' 'stack' ']')?
                    ('[' 'tail_elems' sil-type '*' sil-operand ']')*
                    sil-type

%1 = alloc_ref [stack] $T
%1 = alloc_ref [tail_elems $E * %2 : Builtin.Word] $T
// $T must be a reference type
// %1 has type $T
// $E is the type of the tail-allocated elements
// %2 must be of a builtin integer type
```

Allocates an object of reference type `T`. The object will be initialized with retain count 1; its state will be otherwise uninitialized. The optional `objc` attribute indicates that the object should be allocated using Objective-C's allocation methods (`+allocWithZone:`).

The optional `stack` attribute indicates that the object can be allocated on the stack instead on the heap. In this case the instruction must be balanced with a `dealloc_stack_ref` instruction to mark the end of the object's lifetime. Note that the `stack` attribute only specifies that stack allocation is possible. The final decision on stack allocation is done during llvm IR generation. This is because the decision also depends on the object size, which is not necessarily known at SIL level.

The optional `tail_elems` attributes specifies the amount of space to be reserved for tail-allocated arrays of given element types and element counts. If there are more than one `tail_elems` attributes then the tail arrays are allocated in the specified order. The count-operand must be of a builtin integer type. The instructions `ref_tail_addr` and `tail_addr` can be used to project the tail elements.

The `objc` attribute cannot be used together with `tail_elems`.

alloc_ref_dynamic

```
sil-instruction ::= 'alloc_ref_dynamic'
                    ('[' 'objc' ']')?
                    ('[' 'tail_elems' sil-type '*' sil-operand ']')*
                    sil-operand ',' sil-type

%1 = alloc_ref_dynamic %0 : $@thick T.Type, $T
%1 = alloc_ref_dynamic [objc] %0 : $@objc_metatype T.Type, $T
%1 = alloc_ref_dynamic [tail_elems $E * %2 : Builtin.Word] %0 : $@thick T.Type, $T
// $T must be a class type
// %1 has type $T
// $E is the type of the tail-allocated elements
// %2 must be of a builtin integer type
```

Allocates an object of class type `T` or a subclass thereof. The dynamic type of the resulting object is specified via the metatype value `%0`. The object will be initialized with retain count 1; its state will be otherwise uninitialized.

The optional `tail_elems` and `objc` attributes have the same effect as for `alloc_ref`. See `alloc_ref` for details.

alloc_box

```
sil-instruction ::= 'alloc_box' sil-type (',' debug-var-attr)*

%1 = alloc_box $T
//   %1 has type $@box T
```

Allocates a reference-counted `@box` on the heap large enough to hold a value of type `T`, along with a retain count and any other metadata required by the runtime. The result of the instruction is the reference-counted `@box` reference that owns the box. The `project_box` instruction is used to retrieve the address of the value inside the box.

The box will be initialized with a retain count of 1; the storage will be uninitialized. The box owns the contained value, and releasing it to a retain count of zero destroys the contained value as if by `destroy_addr`. Releasing a box is undefined behavior if the box's value is uninitialized. To deallocate a box whose value has not been initialized, `dealloc_box` should be used.

alloc_global

```
sil-instruction ::= 'alloc_global' sil-global-name

alloc_global @foo
```

Initialize the storage for a global variable. This instruction has undefined behavior if the global variable has already been initialized.

The type operand must be a lowered object type.

get_async_continuation

```
sil-instruction ::= 'get_async_continuation' '[throws]'? sil-type

%0 = get_async_continuation $T
%0 = get_async_continuation [throws] $U
```

Begins a suspension of an `@async` function. This instruction can only be used inside an `@async` function. The result of the instruction is an `UnsafeContinuation<T>` value, where `T` is the formal type argument to the instruction, or an `UnsafeThrowingContinuation<T>` if the instruction carries the `[throws]` attribute. `T` must be a loadable type. The continuation must be consumed by a `await_async_continuation` terminator on all paths. Between `get_async_continuation` and `await_async_continuation`, the following restrictions apply:

- The function cannot `return`, `throw`, `yield`, or `unwind`.
- There cannot be nested suspend points; namely, the function cannot call another `@async` function, nor can it initiate another suspend point with `get_async_continuation`.

The function suspends execution when the matching `await_async_continuation` terminator is reached, and resumes execution when the continuation is resumed. The continuation resumption operation takes a value of type `T` which is passed back into the function when it resumes execution in the `await_async_continuation` instruction's `resume` successor block. If the instruction has the `[throws]` attribute, it can also be resumed in an error state, in which case the matching `await_async_continuation` instruction must also have an `error` successor.

Within the enclosing SIL function, the result continuation is consumed by the `await_async_continuation`, and cannot be referenced after the `await_async_continuation` executes. Dynamically, the continuation value must be resumed exactly once in the course of the program's execution; it is undefined behavior to resume the continuation more than once. Conversely, failing to resume the continuation will leave the suspended async coroutine hung in its suspended state, leaking any resources it may be holding.

get_async_continuation_addr

```
sil-instruction ::= 'get_async_continuation_addr' '[throws]'? sil-type ',' sil-operand

%1 = get_async_continuation_addr $T, %0 : $*T
%1 = get_async_continuation_addr [throws] $U, %0 : $*U
```

Begins a suspension of an `@async` function, like `get_async_continuation`, additionally binding a specific memory location for receiving the value when the result continuation is resumed. The operand must be an address whose type is the maximally-abstracted lowered type of the formal resume type. The memory must be uninitialized, and must remain allocated until the matching `await_async_continuation` instruction(s) consuming the result continuation have executed. The behavior is otherwise the same as `get_async_continuation`, and the same restrictions apply on code appearing between `get_async_continuation_addr` and `await_async_continuation` as apply between `get_async_continuation` and `await_async_continuation`. Additionally, the state of the memory referenced by the operand is indefinite between the execution of `get_async_continuation_addr` and `await_async_continuation`, and it is undefined behavior to read or modify the memory during this time. After the `await_async_continuation` resumes normally to its `resume` successor, the memory referenced by the operand is initialized with the resume value, and that value is then owned by the current function. If `await_async_continuation` instead resumes to its `error` successor, then the memory remains uninitialized.

hop_to_executor

```
sil-instruction ::= 'hop_to_executor' sil-operand

hop_to_executor %0 : $T

// $T must be Builtin.Executor or conform to the Actor protocol
```

Ensures that all instructions, which need to run on the actor's executor actually run on that executor. This instruction can only be used inside an `@async` function.

Checks if the current executor is the one which is bound to the operand actor. If not, begins a suspension point and enqueues the continuation to the executor which is bound to the operand actor.

SIL generation emits this instruction with operands of actor type as well as of type `Builtin.Executor`. The former are expected to be lowered by the SIL pipeline, so that IR generation only operands of type `Builtin.Executor` remain.

The operand is a guaranteed operand, i.e. not consumed.

extract_executor

```
sil-instruction ::= 'extract_executor' sil-operand

%1 = extract_executor %0 : $T
// $T must be Builtin.Executor or conform to the Actor protocol
// %1 will be of type Builtin.Executor
```

Extracts the executor from the executor or actor operand. SIL generation emits this instruction to produce executor values when

needed (e.g., to provide to a runtime function). It will be lowered away by the SIL pipeline.

The operand is a guaranteed operand, i.e. not consumed.

dealloc_stack

```
sil-instruction ::= 'dealloc_stack' sil-operand

dealloc_stack %0 : $*T
// %0 must be of $*T type
```

Deallocates memory previously allocated by `alloc_stack`. The allocated value in memory must be uninitialized or destroyed prior to being deallocated. This instruction marks the end of the lifetime for the value created by the corresponding `alloc_stack` instruction. The operand must be the shallowest live `alloc_stack` allocation preceding the deallocation. In other words, deallocations must be in last-in, first-out stack order.

dealloc_box

```
sil-instruction ::= 'dealloc_box' sil-operand

dealloc_box %0 : $@box T
```

Deallocates a box, bypassing the reference counting mechanism. The box variable must have a retain count of one. The boxed type must match the type passed to the corresponding `alloc_box` exactly, or else undefined behavior results.

This does not destroy the boxed value. The contents of the value must have been fully uninitialized or destroyed before `dealloc_box` is applied.

project_box

```
sil-instruction ::= 'project_box' sil-operand

%1 = project_box %0 : $@box T

// %1 has type $*T
```

Given a `@box T` reference, produces the address of the value inside the box.

dealloc_stack_ref

```
sil-instruction ::= 'dealloc_stack_ref' sil-operand

dealloc_stack_ref %0 : $T
// $T must be a class type
// %0 must be an 'alloc_ref [stack]' instruction
```

Marks the deallocation of the stack space for an `alloc_ref [stack]`.

dealloc_ref

```
sil-instruction ::= 'dealloc_ref' sil-operand

dealloc_ref %0 : $T
// $T must be a class type
```

Deallocates an uninitialized class type instance, bypassing the reference counting mechanism.

The type of the operand must match the allocated type exactly, or else undefined behavior results.

The instance must have a retain count of one.

This does not destroy stored properties of the instance. The contents of stored properties must be fully uninitialized at the time `dealloc_ref` is applied.

The `stack` attribute indicates that the instruction is the balanced deallocation of its operand which must be a `alloc_ref [stack]`. In this case the instruction marks the end of the object's lifetime but has no other effect.

dealloc_partial_ref

```
sil-instruction ::= 'dealloc_partial_ref' sil-operand sil-metatype

dealloc_partial_ref %0 : $T, %1 : $U.Type
// $T must be a class type
// $T must be a subclass of U
```

Deallocates a partially-initialized class type instance, bypassing the reference counting mechanism.

The type of the operand must be a supertype of the allocated type, or else undefined behavior results.

The instance must have a retain count of one.

All stored properties in classes more derived than the given metatype value must be initialized, and all other stored properties must be uninitialized. The initialized stored properties are destroyed before deallocating the memory for the instance.

This does not destroy the reference type instance. The contents of the heap object must have been fully uninitialized or destroyed before `dealloc_ref` is applied.

Debug Information

Debug information is generally associated with allocations (alloc_stack or alloc_box) by having a Decl node attached to the allocation with a SILLocation. For declarations that have no allocation we have explicit instructions for doing this. This is used by 'let' declarations, which bind a value to a name and for var decls who are promoted into registers. The decl they refer to is attached to the instruction with a SILLocation.

debug_value

```
sil-instruction ::= debug_value '[poison]'? '[moved]'? sil-operand (',' debug-var-attr)* advanced-debug-var-attr* (',' 'expr' debug-info-

debug_value %1 : $Int
```

This indicates that the value of a declaration has changed value to the specified operand. The declaration in question is identified by either the SILLocation attached to the debug_value instruction or the SILLocation specified in the advanced debug variable attributes.

If the '[moved]' flag is set, then one knows that the debug_value's operand is moved at some point of the program, so one can not model the debug_value using constructs that assume that the value is live for the entire function (e.x.: llvm.dbg.declare).

```
debug-var-attr ::= 'var'
debug-var-attr ::= 'let'
debug-var-attr ::= 'name' string-literal
debug-var-attr ::= 'argno' integer-literal
debug-var-attr ::= 'implicit'
```

There are a number of attributes that provide details about the source variable that is being described, including the name of the variable. For function and closure arguments `argno` is the number of the function argument starting with 1. A compiler-generated source variable will be marked `implicit` and optimizers are free to remove it even in -Onone.

If the '[poison]' flag is set, then all references within this debug value will be overwritten with a sentinel at this point in the program. This is used in debug builds when shortening non-trivial value lifetimes to ensure the debugger cannot inspect invalid memory. `debug_value` instructions with the poison flag are not generated until OSSA is lowered. They are not expected to be serialized within the module, and the pipeline is not expected to do any significant code motion after lowering.

```
advanced-debug-var-attr ::= '(' 'name' string-literal (',' sil-instruction-source-info)? ')'
advanced-debug-var-attr ::= 'type' sil-type
```

Advanced debug variable attributes represent source locations and the type of the source variable when it was originally declared. It is useful when we're indirectly associating the SSA value with the source variable (via SIL DIExpression, for example) in which case SSA value's type is different from that of source variable.

```
debug-info-expr   ::= di-expr-operand (':' di-expr-operand)*
di-expr-operand   ::= di-expr-operator (':' sil-operand)*
di-expr-operator  ::= 'op_fragment'
di-expr-operator  ::= 'op_deref'
```

SIL debug info expression (SIL DIExpression) is a powerful method to connect SSA value with the source variable in an indirect fashion. Di-expression in SIL uses a stack based execution model to evaluate the expression and apply on the associated (SIL) SSA value before connecting it with the debug variable. For instance, given the following SIL code:

```
debug_value %a : $*Int, name "x", expr op_deref
```

It means: "You can get the value of source variable 'x' by *dereferencing* SSA value `%a`". The `op_deref` is a SIL DIExpression operator that represents "dereference". If there are multiple SIL DIExpression operators (or arguments), they are evaluated from left to right:

```
debug_value %b : $**Int, name "y", expr op_deref:op_deref
```

In the snippet above, two `op_deref` operators will be applied on SSA value `%b` sequentially.

Note that normally when the SSA value has an address type, there will be a `op_deref` in the SIL DIExpression. Because there is no pointer in Swift so you always need to dereference an address-type SSA value to get the value of a source variable. However, if the SSA value is a `alloc_stack`, the `debug_value` is used to indicate the *declaration* of a source variable. Or, you can say, used to specify the location (memory address) of the source variable. Therefore, we don't need to add a `op_deref` in this case:

```
%a = alloc_stack $Int, ...
debug_value %a : $*Int, name "my_var"
```

The `op_fragment` operator is used to specify the SSA value of a specific field in an aggregate-type source variable. This SIL DIExpression operator takes a field declaration -- which references the desired sub-field in source variable -- as its argument. Here is an example:

```
struct MyStruct {
  var x: Int
  var y: Int
}
...
debug_value %1 : $Int, var, (name "the_struct", loc "file.swift":8:7), type $MyStruct, expr op_fragment:#MyStruct.y, loc "file.swift":9:4
```

In the snippet above, source variable "the_struct" has an aggregate type `$MyStruct` and we use a SIL DIExpression with `op_fragment` operator to associate `%1` to the `y` member variable (via the `#MyStruct.y` directive) inside "the_struct". Note that the extra source location directive follows rigt after `name "the_struct"` indicate that "the_struct" was originally declared in line 8, but not until line 9 -- the current `debug_value` instruction's source location -- does member `y` got updated with SSA value `%1`.

It is worth noting that a SIL DIExpression is similar to !DIExpression in LLVM debug info metadata. While LLVM represents `!DIExpression` are a list of 64-bit integers, SIL DIExpression can have elements with various types, like AST nodes or strings.

Accessing Memory

load

```
sil-instruction ::= 'load' sil-operand

%1 = load %0 : $*T
// %0 must be of a $*T address type for loadable type $T
// %1 will be of type $T
```

Loads the value at address `%0` from memory. `T` must be a loadable type. This does not affect the reference count, if any, of the loaded value; the value must be retained explicitly if necessary. It is undefined behavior to load from uninitialized memory or to load from an address that points to deallocated storage.

store

```
sil-instruction ::= 'store' sil-value 'to' sil-operand

store %0 to %1 : $*T
// $T must be a loadable type
```

Stores the value `%0` to memory at address `%1`. The type of %1 is `*T` and the type of `%0` is `T`, which must be a loadable type. This will overwrite the memory at `%1`. If `%1` already references a value that requires `release` or other cleanup, that value must be loaded before being stored over and cleaned up. It is undefined behavior to store to an address that points to deallocated storage.

load_borrow

```
sil-instruction ::= 'load_borrow' sil-value

%1 = load_borrow %0 : $*T
// $T must be a loadable type
```

Loads the value `%1` from the memory location `%0`. The load_borrow instruction creates a borrowed scope in which a read-only borrow value `%1` can be used to read the value stored in `%0`. The end of scope is delimited by an end_borrow instruction. All load_borrow instructions must be paired with exactly one end_borrow instruction along any path through the program. Until end_borrow, it is illegal to invalidate or store to `%0`.

store_borrow

```
sil-instruction ::= 'store_borrow' sil-value 'to' sil-operand

store_borrow %0 to %1 : $*T
// $T must be a loadable type
// %1 must be an alloc_stack $T
```

Stores the value `%0` to a stack location `%1`, which must be an `alloc_stack $T`. The stored value is alive until the `dealloc_stack` or until another `store_borrow` overwrites the value. During the its lifetime, the stored value must not be modified or destroyed. The source value `%0` is borrowed (i.e. not copied) and it's borrow scope must outlive the lifetime of the stored value.

Note: This is the current implementation and the design is not final.

begin_borrow

```
sil-instruction ::= 'begin_borrow' '[lexical]'? sil-operand

%1 = begin_borrow %0 : $T
```

Given a value `%0` with Owned or Guaranteed ownership, produces a new same typed value with Guaranteed ownership: `%1`. `%1` is guaranteed to have a lifetime ending use (e.x.: end_borrow) along all paths that do not end in Dead End Blocks. This begin_borrow and the lifetime ending uses of `%1` are considered to be liveness requiring uses of `%0` and as such in the region in between this borrow and its lifetime ending use, `%0` must be live. This makes sense semantically since `%1` is modeling a new value with a dependent lifetime on `%0`.

The optional `lexical` attribute specifies that the operand corresponds to a local variable in the Swift source, so special care must be taken when moving the end_borrow.

This instruction is only valid in functions in Ownership SSA form.

end_borrow

```
sil-instruction ::= 'end_borrow' sil-operand
```

```
// somewhere earlier
// %1 = begin_borrow %0
end_borrow %1 : $T
```

Ends the scope for which the Guaranteed ownership possessing SILValue `%1` is borrowed from the SILValue `%0`. Must be paired with at most 1 borrowing instruction (like load_borrow, begin_borrow) along any path through the program. In the region in between the borrow instruction and the end_borrow, the original SILValue can not be modified. This means that:

1. If `%0` is an address, `%0` can not be written to.
2. If `%0` is a non-trivial value, `%0` can not be destroyed.

We require that `%1` and `%0` have the same type ignoring SILValueCategory.

This instruction is only valid in functions in Ownership SSA form.

end_lifetime

```
sil-instruction ::= 'end_lifetime' sil-operand
```

This instruction signifies the end of it's operand's lifetime to the ownership verifier. It is inserted by the compiler in instances where it could be illegal to insert a destroy operation. Ex: if the sil-operand had an undef value.

This instruction is valid only in OSSA and is lowered to a no-op when lowering to non-OSSA.

assign

```
sil-instruction ::= 'assign' sil-value 'to' sil-operand

assign %0 to %1 : $*T
// $T must be a loadable type
```

Represents an abstract assignment of the value `%0` to memory at address `%1` without specifying whether it is an initialization or a normal store. The type of %1 is `*T` and the type of `%0` is `T`, which must be a loadable type. This will overwrite the memory at `%1` and destroy the value currently held there.

The purpose of the assign instruction is to simplify the definitive initialization analysis on loadable variables by removing what would otherwise appear to be a load and use of the current value. It is produced by SILGen, which cannot know which assignments are meant to be initializations. If it is deemed to be an initialization, it can be replaced with a store; otherwise, it must be replaced with a sequence that also correctly destroys the current value.

This instruction is only valid in Raw SIL and is rewritten as appropriate by the definitive initialization pass.

assign_by_wrapper

```
sil-instruction ::= 'assign_by_wrapper' sil-operand 'to' mode? sil-operand ',' 'init' sil-operand ',' 'set' sil-operand

mode ::= '[initialization]' | '[assign]' | '[assign_wrapped_value]'

assign_by_wrapper %0 : $S to %1 : $*T, init %2 : $F, set %3 : $G
// $S can be a value or address type
// $T must be the type of a property wrapper.
// $F must be a function type, taking $S as a single argument (or multiple arguments in case of a tuple) and returning $T
// $G must be a function type, taking $S as a single argument (or multiple arguments in case of a tuple) and without a return value
```

Similar to the assign instruction, but the assignment is done via a delegate.

Initially the instruction is created with no mode. Once the mode is decided (by the definitive initialization pass), the instruction is lowered as follows:

If the mode is `initialization`, the function `%2` is called with `%0` as argument. The result is stored to `%1`. In case of an address type, `%1` is simply passed as a first out-argument to `%2`.

The `assign` mode works similar to `initialization`, except that the destination is "assigned" rather than "initialized". This means that the existing value in the destination is destroyed before the new value is stored.

If the mode is `assign_wrapped_value`, the function `%3` is called with `%0` as argument. As `%3` is a setter (e.g. for the property in the containing nominal type), the destination address `%1` is not used in this case.

This instruction is only valid in Raw SIL and is rewritten as appropriate by the definitive initialization pass.

mark_uninitialized

```
sil-instruction ::= 'mark_uninitialized' '[' mu_kind ']' sil-operand
mu_kind ::= 'var'
mu_kind ::= 'rootself'
mu_kind ::= 'crossmodulerootself'
mu_kind ::= 'derivedself'
mu_kind ::= 'derivedselfonly'
mu_kind ::= 'delegatingself'
mu_kind ::= 'delegatingselfallocated'

%2 = mark_uninitialized [var] %1 : $*T
// $T must be an address
```

Indicates that a symbolic memory location is uninitialized, and must be explicitly initialized before it escapes or before the current function returns. This instruction returns its operands, and all accesses within the function must be performed against the return value of the mark_uninitialized instruction.

The kind of mark_uninitialized instruction specifies the type of data the mark_uninitialized instruction refers to:

- `var`: designates the start of a normal variable live range
- `rootself`: designates `self` in a struct, enum, or root class
- `crossmodulerootself`: same as `rootself`, but in a case where it's not really safe to treat `self` as a root because the original module might add more stored properties. This is only used for Swift 4 compatibility.
- `derivedself`: designates `self` in a derived (non-root) class
- `derivedselfonly`: designates `self` in a derived (non-root) class whose stored properties have already been initialized
- `delegatingself`: designates `self` on a struct, enum, or class in a delegating constructor (one that calls self.init)
- `delegatingselfallocated`: designates `self` on a class convenience initializer's initializing entry point

The purpose of the `mark_uninitialized` instruction is to enable definitive initialization analysis for global variables (when marked as 'globalvar') and instance variables (when marked as 'rootinit'), which need to be distinguished from simple allocations.

It is produced by SILGen, and is only valid in Raw SIL. It is rewritten as appropriate by the definitive initialization pass.

mark_function_escape

```
sil-instruction ::= 'mark_function_escape' sil-operand (',' sil-operand)

mark_function_escape %1 : $*T
```

Indicates that a function definition closes over a symbolic memory location. This instruction is variadic, and all of its operands must be addresses.

The purpose of the `mark_function_escape` instruction is to enable definitive initialization analysis for global variables and instance variables, which are not represented as box allocations.

It is produced by SILGen, and is only valid in Raw SIL. It is rewritten as appropriate by the definitive initialization pass.

mark_uninitialized_behavior

```
init-case ::= sil-value sil-apply-substitution-list? '(' sil-value ')' ':' sil-type
set-case ::= sil-value sil-apply-substitution-list? '(' sil-value ')' ':' sil-type
sil-instruction ::= 'mark_uninitialized_behavior' init-case set-case
```

```
mark_uninitialized_behavior %init<Subs>(%storage) : $T -> U,
                            %set<Subs>(%self) : $V -> W
```

Indicates that a logical property is uninitialized at this point and needs to be initialized by the end of the function and before any escape point for this instruction. Assignments to the property trigger the behavior's `init` or `set` logic based on the logical initialization state of the property.

It is expected that the `init-case` is passed some sort of storage and the `set` case is passed `self`.

This is only valid in Raw SIL.

copy_addr

```
sil-instruction ::= 'copy_addr' '[take]'? sil-value
                    'to' '[initialization]'? sil-operand

copy_addr [take] %0 to [initialization] %1 : $*T
// %0 and %1 must be of the same $*T address type
```

Loads the value at address `%0` from memory and assigns a copy of it back into memory at address `%1`. A bare `copy_addr` instruction when `T` is a non-trivial type:

```
copy_addr %0 to %1 : $*T
```

is equivalent to:

```
%new = load %0 : $*T        // Load the new value from the source
%old = load %1 : $*T        // Load the old value from the destination
strong_retain %new : $T           // Retain the new value
strong_release %old : $T          // Release the old
store %new to %1 : $*T      // Store the new value to the destination
```

except that `copy_addr` may be used even if `%0` is of an address-only type. The `copy_addr` may be given one or both of the `[take]` or `[initialization]` attributes:

- `[take]` destroys the value at the source address in the course of the copy.
- `[initialization]` indicates that the destination address is uninitialized. Without the attribute, the destination address is treated as already initialized, and the existing value will be destroyed before the new value is stored.

The three attributed forms thus behave like the following loadable type operations:

```
// take-assignment
  copy_addr [take] %0 to %1 : $*T
// is equivalent to:
  %new = load %0 : $*T
  %old = load %1 : $*T
  // no retain of %new!
  strong_release %old : $T
  store %new to %1 : $*T

// copy-initialization
  copy_addr %0 to [initialization] %1 : $*T
// is equivalent to:
  %new = load %0 : $*T
  strong_retain %new : $T
  // no load/release of %old!
  store %new to %1 : $*T

// take-initialization
  copy_addr [take] %0 to [initialization] %1 : $*T
// is equivalent to:
  %new = load %0 : $*T
  // no retain of %new!
  // no load/release of %old!
  store %new to %1 : $*T
```

If `T` is a trivial type, then `copy_addr` is always equivalent to its take-initialization form.

destroy_addr

```
sil-instruction ::= 'destroy_addr' sil-operand

destroy_addr %0 : $*T
// %0 must be of an address $*T type
```

Destroys the value in memory at address `%0`. If `T` is a non-trivial type, This is equivalent to:

```
%1 = load %0
strong_release %1
```

except that `destroy_addr` may be used even if `%0` is of an address-only type. This does not deallocate memory; it only destroys the pointed-to value, leaving the memory uninitialized.

If `T` is a trivial type, then `destroy_addr` can be safely eliminated. However, a memory location `%a` must not be accessed after `destroy_addr %a` (which has not yet been eliminated) regardless of its type.

index_addr

```
sil-instruction ::= 'index_addr' sil-operand ',' sil-operand

%2 = index_addr %0 : $*T, %1 : $Builtin.Int<n>
// %0 must be of an address type $*T
// %1 must be of a builtin integer type
// %2 will be of type $*T
```

Given an address that references into an array of values, returns the address of the `%1`-th element relative to `%0`. The address must reference into a contiguous array. It is undefined to try to reference offsets within a non-array value, such as fields within a homogeneous struct or tuple type, or bytes within a value, using `index_addr`. (`Int8` address types have no special behavior in this regard, unlike `char*` or `void*` in C.) It is also undefined behavior to index out of bounds of an array, except to index the "past-the-end" address of the array.

tail_addr

```
sil-instruction ::= 'tail_addr' sil-operand ',' sil-operand ',' sil-type

%2 = tail_addr %0 : $*T, %1 : $Builtin.Int<n>, $E
// %0 must be of an address type $*T
// %1 must be of a builtin integer type
// %2 will be of type $*E
```

Given an address of an array of `%1` values, returns the address of an element which is tail-allocated after the array. This instruction is equivalent to `index_addr` except that the resulting address is aligned-up to the tail-element type `$E`.

This instruction is used to project the N-th tail-allocated array from an object which is created by an `alloc_ref` with multiple `tail_elems`. The first operand is the address of an element of the (N-1)-th array, usually the first element. The second operand is the number of elements until the end of that array. The result is the address of the first element of the N-th array.

It is undefined behavior if the provided address, count and type do not match the actual layout of tail-allocated arrays of the underlying object.

index_raw_pointer

```
sil-instruction ::= 'index_raw_pointer' sil-operand ',' sil-operand

%2 = index_raw_pointer %0 : $Builtin.RawPointer, %1 : $Builtin.Int<n>
// %0 must be of $Builtin.RawPointer type
```

```
// %1 must be of a builtin integer type
// %2 will be of type $Builtin.RawPointer
```

Given a `Builtin.RawPointer` value `%0`, returns a pointer value at the byte offset `%1` relative to `%0`.

bind_memory

```
sil-instruction ::= 'bind_memory' sil-operand ',' sil-operand 'to' sil-type

%token = bind_memory %0 : $Builtin.RawPointer, %1 : $Builtin.Word to $T
// %0 must be of $Builtin.RawPointer type
// %1 must be of $Builtin.Word type
// %token is an opaque $Builtin.Word representing the previously bound types
// for this memory region.
```

Binds memory at `Builtin.RawPointer` value `%0` to type `$T` with enough capacity to hold `%1` values. See SE-0107: UnsafeRawPointer.

Produces a opaque token representing the previous memory state. For memory binding semantics, this state includes the type that the memory was previously bound to. The token cannot, however, be used to retrieve a metatype. It's value is only meaningful to the Swift runtime for typed pointer verification.

rebind_memory

```
sil-instruction ::= 'rebind_memory' sil-operand ' 'to' sil-value

%out_token = rebind_memory %0 : $Builtin.RawPointer to %in_token
// %0 must be of $Builtin.RawPointer type
// %in_token represents a cached set of bound types from a prior memory state.
// %out_token is an opaque $Builtin.Word representing the previously bound
// types for this memory region.
```

This instruction's semantics are identical to `bind_memory`, except that the types to which memory will be bound, and the extent of the memory region is unknown at compile time. Instead, the bound-types are represented by a token that was produced by a prior memory binding operation. `%in_token` must be the result of bind_memory or rebind_memory.

begin_access

```
sil-instruction ::= 'begin_access' '[' sil-access ']' '[' sil-enforcement ']' '[no_nested_conflict]'? '[builtin]'? sil-operand ':' sil-ty
sil-access ::= init
sil-access ::= read
sil-access ::= modify
sil-access ::= deinit
sil-enforcement ::= unknown
sil-enforcement ::= static
sil-enforcement ::= dynamic
sil-enforcement ::= unsafe
%1 = begin_access [read] [unknown] %0 : $*T
// %0 must be of $*T type.
```

Begins an access to the target memory.

The operand must be a *root address derivation*:

- a function argument,
- an `alloc_stack` instruction,
- a `project_box` instruction,
- a `global_addr` instruction,
- a `ref_element_addr` instruction, or
- another `begin_access` instruction.

It will eventually become a basic structural rule of SIL that no memory access instructions can be directly applied to the result of one of these instructions; they can only be applied to the result of a `begin_access` on them. For now, this rule will be conditional based on compiler settings and the SIL stage.

An access is ended with a corresponding `end_access`. Accesses must be uniquely ended on every control flow path which leads to either a function exit or back to the `begin_access` instruction. The set of active accesses must be the same on every edge into a basic block.

An `init` access takes uninitialized memory and initializes it. It must always use `static` enforcement.

An `deinit` access takes initialized memory and leaves it uninitialized. It must always use `static` enforcement.

`read` and `modify` accesses take initialized memory and leave it initialized. They may use `unknown` enforcement only in the `raw` SIL stage.

A `no_nested_conflict` access has no potentially conflicting access within its scope (on any control flow path between it and its corresponding `end_access`). Consequently, the access will not need to be tracked by the runtime for the duration of its scope. This access may still conflict with an outer access scope; therefore may still require dynamic enforcement at a single point.

A `builtin` access was emitted for a user-controlled Builtin (e.g. the standard library's KeyPath access). Non-builtin accesses are auto-generated by the compiler to enforce formal access that derives from the language. A `builtin` access is always fully enforced regardless of the compilation mode because it may be used to enforce access outside of the current module.

end_access

```
sil-instruction ::= 'end_access' ( '[' 'abort' ']' )? sil-operand
```

Ends an access. The operand must be a `begin_access` instruction.

If the `begin_access` is `init` or `deinit`, the `end_access` may be an `abort`, indicating that the described transition did not in fact take place.

begin_unpaired_access

```
sil-instruction ::= 'begin_unpaired_access' '[' sil-access ']' '[' sil-enforcement ']' '[no_nested_conflict]'? '[builtin]'? sil-operand :
sil-access ::= init
sil-access ::= read
sil-access ::= modify
sil-access ::= deinit
sil-enforcement ::= unknown
sil-enforcement ::= static
sil-enforcement ::= dynamic
sil-enforcement ::= unsafe
%2 = begin_unpaired_access [read] [dynamic] %0 : $*T, %1 : $*Builtin.UnsafeValueBuffer
// %0 must be of $*T type.
```

Begins an access to the target memory. This has the same semantics and obeys all the same constraints as `begin_access`. With the following exceptions:

- `begin_unpaired_access` has an additional operand for the scratch buffer used to uniquely identify this access within its scope.
- An access initiated by `begin_unpaired_access` must end with `end_unpaired_access` unless it has the `no_nested_conflict` flag. A `begin_unpaired_access` with `no_nested_conflict` is effectively an instantaneous access with no associated scope.
- The associated `end_unpaired_access` must use the same scratch buffer.

end_unpaired_access

```
sil-instruction ::= 'end_unpaired_access' ( '[' 'abort' ']' )? '[' sil-enforcement ']' sil-operand : $*Builtin.UnsafeValueBuffer
sil-enforcement ::= unknown
sil-enforcement ::= static
sil-enforcement ::= dynamic
sil-enforcement ::= unsafe
```

```
%1 = end_unpaired_access [dynamic] %0 : $*Builtin.UnsafeValueBuffer
```

Ends an access. This has the same semantics and constraints as `end_access` with the following exceptions:

- The single operand refers to the scratch buffer that uniquely identified the access with this scope.
- The enforcement level is reiterated, since the corresponding `begin_unpaired_access` may not be statically discoverable. It must be identical to the `begin_unpaired_access` enforcement.

Reference Counting

These instructions handle reference counting of heap objects. Values of strong reference type have ownership semantics for the referenced heap object. Retain and release operations, however, are never implicit in SIL and always must be explicitly performed where needed. Retains and releases on the value may be freely moved, and balancing retains and releases may be deleted, so long as an owning retain count is maintained for the uses of the value.

All reference-counting operations are defined to work correctly on null references (whether strong, unowned, or weak). A non-null reference must actually refer to a valid object of the indicated type (or a subtype). Address operands are required to be valid and non-null.

While SIL makes reference-counting operations explicit, the SIL type system also fully represents strength of reference. This is useful for several reasons:

1. Type-safety: it is impossible to erroneously emit SIL that naively uses a `@weak` or `@unowned` reference as if it were a strong reference.
2. Consistency: when a reference is kept in memory, instructions like `copy_addr` and `destroy_addr` implicitly carry the right semantics in the type of the address, rather than needing special variants or flags.
3. Ease of tooling: SIL directly stores the user's intended strength of reference, making it straightforward to generate instrumentation that would convey this to a memory profiler. In principle, with only a modest number of additions and restrictions on SIL, it would even be possible to drop all reference-counting instructions and use the type information to feed a garbage collector.

strong_retain

```
sil-instruction ::= 'strong_retain' sil-operand

strong_retain %0 : $T
// $T must be a reference type
```

Increases the strong retain count of the heap object referenced by `%0`.

strong_release

```
strong_release %0 : $T
// $T must be a reference type.
```

Decrements the strong reference count of the heap object referenced by `%0`. If the release operation brings the strong reference count of the object to zero, the object is destroyed and `@weak` references are cleared. When both its strong and unowned reference counts reach zero, the object's memory is deallocated.

set_deallocating

```
set_deallocating %0 : $T
// $T must be a reference type.
```

Explicitly sets the state of the object referenced by `%0` to deallocated. This is the same operation what's done by a strong_release immediately before it calls the deallocator of the object.

It is expected that the strong reference count of the object is one. Furthermore, no other thread may increment the strong reference count during execution of this instruction.

strong_copy_unowned_value

```
sil-instruction ::= 'strong_copy_unowned_value' sil-operand

%1 = strong_copy_unowned_value %0 : $@unowned T
// %1 will be a strong @owned value of type $T.
// $T must be a reference type
```

Asserts that the strong reference count of the heap object referenced by `%0` is still positive, then increments the reference count and returns a new strong reference to `%0`. The intention is that this instruction is used as a "safe ownership conversion" from `unowned` to `strong`.

strong_retain_unowned

```
sil-instruction ::= 'strong_retain_unowned' sil-operand

strong_retain_unowned %0 : $@unowned T
// $T must be a reference type
```

Asserts that the strong reference count of the heap object referenced by `%0` is still positive, then increases it by one.

unowned_retain

```
sil-instruction ::= 'unowned_retain' sil-operand

unowned_retain %0 : $@unowned T
// $T must be a reference type
```

Increments the unowned reference count of the heap object underlying `%0`.

unowned_release

```
sil-instruction ::= 'unowned_release' sil-operand

unowned_release %0 : $@unowned T
// $T must be a reference type
```

Decrements the unowned reference count of the heap object referenced by `%0`. When both its strong and unowned reference counts reach zero, the object's memory is deallocated.

load_weak

```
sil-instruction ::= 'load_weak' '[take]'? sil-operand

load_weak [take] %0 : $*@sil_weak Optional<T>
// $T must be an optional wrapping a reference type
```

Increments the strong reference count of the heap object held in the operand, which must be an initialized weak reference. The result is value of type `$Optional<T>`, except that it is `null` if the heap object has begun deallocation.

If `[take]` is specified then the underlying weak reference is invalidated implying that the weak reference count of the loaded value is decremented. If `[take]` is not specified then the underlying weak reference count is not affected by this operation (i.e. it is a +0 weak ref count operation). In either case, the strong reference count will be incremented before any changes to the weak reference count.

This operation must be atomic with respect to the final `strong_release` on the operand heap object. It need not be atomic with respect to `store_weak` operations on the same address.

store_weak

```
sil-instruction ::= 'store_weak' sil-value 'to' '[initialization]'? sil-operand
```

```
store_weak %0 to [initialization] %1 : $*@sil_weak Optional<T>
// $T must be an optional wrapping a reference type
```

Initializes or reassigns a weak reference. The operand may be `nil`.

If `[initialization]` is given, the weak reference must currently either be uninitialized or destroyed. If it is not given, the weak reference must currently be initialized. After the evaluation:

- The value that was originally referenced by the weak reference will have its weak reference count decremented by 1.
- If the optionally typed operand is non-nil, the strong reference wrapped in the optional has its weak reference count incremented by 1. In contrast, the reference's strong reference count is not touched.

This operation must be atomic with respect to the final `strong_release` on the operand (source) heap object. It need not be atomic with respect to `store_weak` or `load_weak` operations on the same address.

load_unowned

TODO: Fill this in

store_unowned

TODO: Fill this in

fix_lifetime

```
sil-instruction :: 'fix_lifetime' sil-operand

fix_lifetime %0 : $T
// Fix the lifetime of a value %0
fix_lifetime %1 : $*T
// Fix the lifetime of the memory object referenced by %1
```

Acts as a use of a value operand, or of the value in memory referenced by an address operand. Optimizations may not move operations that would destroy the value, such as `release_value`, `strong_release`, `copy_addr [take]`, or `destroy_addr`, past this instruction.

mark_dependence

```
sil-instruction :: 'mark_dependence' sil-operand 'on' sil-operand

%2 = mark_dependence %0 : $*T on %1 : $Builtin.NativeObject
```

Indicates that the validity of the first operand depends on the value of the second operand. Operations that would destroy the second value must not be moved before any instructions which depend on the result of this instruction, exactly as if the address had been obviously derived from that operand (e.g. using `ref_element_addr`).

The result is always equal to the first operand. The first operand will typically be an address, but it could be an address in a non-obvious form, such as a Builtin.RawPointer or a struct containing the same. Transformations should be somewhat forgiving here.

The second operand may have either object or address type. In the latter case, the dependency is on the current value stored in the address.

is_unique

```
sil-instruction ::= 'is_unique' sil-operand

%1 = is_unique %0 : $*T
// $T must be a reference-counted type
// %1 will be of type Builtin.Int1
```

Checks whether %0 is the address of a unique reference to a memory object. Returns 1 if the strong reference count is 1, and 0 if the strong reference count is greater than 1.

A discussion of the semantics can be found here: is_unique instruction

begin_cow_mutation

```
sil-instruction ::= 'begin_cow_mutation' '[native]'? sil-operand

(%1, %2) = begin_cow_mutation %0 : $C
// $C must be a reference-counted type
// %1 will be of type Builtin.Int1
// %2 will be of type C
```

Checks whether %0 is a unique reference to a memory object. Returns 1 in the first result if the strong reference count is 1, and 0 if the strong reference count is greater than 1.

Returns the reference operand in the second result. The returned reference can be used to mutate the object. Technically, the returned reference is the same as the operand. But it's important that optimizations see the result as a different SSA value than the operand. This is important to ensure the correctness of `ref_element_addr [immutable]`.

The operand is consumed and the second result is returned as owned.

The optional `native` attribute specifies that the operand has native Swift reference counting.

end_cow_mutation

```
sil-instruction ::= 'end_cow_mutation' '[keep_unique]'? sil-operand

%1 = end_cow_mutation %0 : $C
// $C must be a reference-counted type
// %1 will be of type C
```

Marks the end of the mutation of a reference counted object. Returns the reference operand. Technically, the returned reference is the same as the operand. But it's important that optimizations see the result as a different SSA value than the operand. This is important to ensure the correctness of `ref_element_addr [immutable]`.

The operand is consumed and the result is returned as owned. The result is guaranteed to be uniquely referenced.

The optional `keep_unique` attribute indicates that the optimizer must not replace this reference with a not uniquely reference object.

is_escaping_closure

```
sil-instruction ::= 'is_escaping_closure' sil-operand

%1 = is_escaping_closure %0 : $@callee_guaranteed () -> ()
// %0 must be an escaping swift closure.
// %1 will be of type Builtin.Int1
```

Checks whether the context reference is not nil and bigger than one and returns true if it is.

copy_block

```
sil-instruction :: 'copy_block' sil-operand

%1 = copy_block %0 : $@convention(block) T -> U
```

Performs a copy of an Objective-C block. Unlike retains of other reference-counted types, this can produce a different value from the operand if the block is copied from the stack to the heap.

copy_block_without_escaping

```
sil-instruction :: 'copy_block_without_escaping' sil-operand 'withoutEscaping' sil-operand
```

```
%1 = copy_block %0 : $@convention(block) T -> U withoutEscaping %1 : $T -> U
```

Performs a copy of an Objective-C block. Unlike retains of other reference-counted types, this can produce a different value from the operand if the block is copied from the stack to the heap.

Additionally, consumes the `withoutEscaping` operand `%1` which is the closure sentinel. SILGen emits these instructions when it passes @noescape swift closures to Objective C. A mandatory SIL pass will lower this instruction into a `copy_block` and a `is_escaping`/`cond_fail`/`destroy_value` at the end of the lifetime of the objective c closure parameter to check whether the sentinel closure was escaped.

builtin "unsafeGuaranteed"

```
sil-instruction := 'builtin' '"unsafeGuaranteed"' '<' sil-type '>' '(' sil-operand')' ':' sil-type

%1 = builtin "unsafeGuaranteed"<T>(%0 : $T) : ($T, Builtin.Int1)
// $T must be of AnyObject type.
```

Asserts that there exists another reference of the value `%0` for the scope delineated by the call of this builtin up to the first call of a `builtin "unsafeGuaranteedEnd"` instruction that uses the second element `%1.1` of the returned value. If no such instruction can be found nothing can be assumed. This assertion holds for uses of the first tuple element of the returned value `%1.0` within this scope. The returned reference value equals the input `%0`.

builtin "unsafeGuaranteedEnd"

```
sil-instruction := 'builtin' '"unsafeGuaranteedEnd"' '(' sil-operand')'

%1 = builtin "unsafeGuaranteedEnd"(%0 : $Builtin.Int1)
// $T must be of AnyObject type.
```

Ends the scope for the `builtin "unsafeGuaranteed"` instruction.

Literals

These instructions bind SIL values to literal constants or to global entities.

function_ref

```
sil-instruction ::= 'function_ref' sil-function-name ':' sil-type

%1 = function_ref @function : $@convention(thin) T -> U
// $@convention(thin) T -> U must be a thin function type
// %1 has type $T -> U
```

Creates a reference to a SIL function.

dynamic_function_ref

```
sil-instruction ::= 'dynamic_function_ref' sil-function-name ':' sil-type

%1 = dynamic_function_ref @function : $@convention(thin) T -> U
// $@convention(thin) T -> U must be a thin function type
// %1 has type $T -> U
```

Creates a reference to a *dynamically_replacable* SIL function. A *dynamically_replacable* SIL function can be replaced at runtime.

For the following Swift code:

```
dynamic func test_dynamically_replaceable() {}

func test_dynamic_call() {
  test_dynamically_replaceable()
}
```

We will generate:

```
sil [dynamically_replacable] @test_dynamically_replaceable : $@convention(thin) () -> () {
bb0:
  %0 = tuple ()
  return %0 : $()
}

sil @test_dynamic_call : $@convention(thin) () -> () {
bb0:
  %0 = dynamic_function_ref @test_dynamically_replaceable : $@convention(thin) () -> ()
  %1 = apply %0() : $@convention(thin) () -> ()
  %2 = tuple ()
  return %2 : $()
}
```

prev_dynamic_function_ref

```
sil-instruction ::= 'prev_dynamic_function_ref' sil-function-name ':' sil-type

%1 = prev_dynamic_function_ref @function : $@convention(thin) T -> U
// $@convention(thin) T -> U must be a thin function type
// %1 has type $T -> U
```

Creates a reference to a previous implementation of a *dynamic_replacement* SIL function.

For the following Swift code:

```
@_dynamicReplacement(for: test_dynamically_replaceable())
func test_replacement() {
  test_dynamically_replaceable() // calls previous implementation
}
```

We will generate:

```
sil [dynamic_replacement_for "test_dynamically_replaceable"] @test_replacement : $@convention(thin) () -> () {
bb0:
  %0 = prev_dynamic_function_ref @test_replacement : $@convention(thin) () -> ()
  %1 = apply %0() : $@convention(thin) () -> ()
  %2 = tuple ()
  return %2 : $()
}
```

global_addr

```
sil-instruction ::= 'global_addr' sil-global-name ':' sil-type

%1 = global_addr @foo : $*Builtin.Word
```

Creates a reference to the address of a global variable which has been previously initialized by `alloc_global`. It is undefined behavior to perform this operation on a global variable which has not been initialized, except the global variable has a static initializer.

global_value

```
sil-instruction ::= 'global_value' sil-global-name ':' sil-type

%1 = global_value @v : $T
```

Returns the value of a global variable which has been previously initialized by `alloc_global`. It is undefined behavior to perform this operation on a global variable which has not been initialized, except the global variable has a static initializer.

integer_literal

```
sil-instruction ::= 'integer_literal' sil-type ',' int-literal

%1 = integer_literal $Builtin.Int<n>, 123
// $Builtin.Int<n> must be a builtin integer type
// %1 has type $Builtin.Int<n>
```

Creates an integer literal value. The result will be of type `Builtin.Int<n>`, which must be a builtin integer type. The literal value is specified using Swift's integer literal syntax.

float_literal

```
sil-instruction ::= 'float_literal' sil-type ',' int-literal

%1 = float_literal $Builtin.FP<n>, 0x3F800000
// $Builtin.FP<n> must be a builtin floating-point type
// %1 has type $Builtin.FP<n>
```

Creates a floating-point literal value. The result will be of type `Builtin.FP<n>`, which must be a builtin floating-point type. The literal value is specified as the bitwise representation of the floating point value, using Swift's hexadecimal integer literal syntax.

string_literal

```
sil-instruction ::= 'string_literal' encoding string-literal
encoding ::= 'utf8'
encoding ::= 'utf16'
encoding ::= 'objc_selector'

%1 = string_literal "asdf"
// %1 has type $Builtin.RawPointer
```

Creates a reference to a string in the global string table. The result is a pointer to the data. The referenced string is always null-terminated. The string literal value is specified using Swift's string literal syntax (though `\()` interpolations are not allowed). When the encoding is `objc_selector`, the string literal produces a reference to a UTF-8-encoded Objective-C selector in the Objective-C method name segment.

base_addr_for_offset

```
sil-instruction ::= 'base_addr_for_offset' sil-type

%1 = base_addr_for_offset $*S
// %1 has type $*S
```

Creates a base address for offset calculations. The result can be used by address projections, like `struct_element_addr`, which themselves return the offset of the projected fields. IR generation simply creates a null pointer for `base_addr_for_offset`.

Dynamic Dispatch

These instructions perform dynamic lookup of class and generic methods.

The `class_method` and `super_method` instructions must reference Swift native methods and always use vtable dispatch.

The `objc_method` and `objc_super_method` instructions must reference Objective-C methods (indicated by the `foreign` marker on a method reference, as in `#NSObject.description!foreign`).

Note that `objc_msgSend` invocations can only be used as the callee of an `apply` instruction or `partial_apply` instruction. They cannot be stored or used as `apply` or `partial_apply` arguments.

class_method

```
sil-instruction ::= 'class_method' sil-method-attributes?
                    sil-operand ',' sil-decl-ref ':' sil-type

%1 = class_method %0 : $T, #T.method : $@convention(class_method) U -> V
// %0 must be of a class type or class metatype $T
// #T.method must be a reference to a Swift native method of T or
// of one of its superclasses
// %1 will be of type $U -> V
```

Looks up a method based on the dynamic type of a class or class metatype instance. It is undefined behavior if the class value is null.

If the static type of the class instance is known, or the method is known to be final, then the instruction is a candidate for devirtualization optimization. A devirtualization pass can consult the module's VTables to find the SIL function that implements the method and promote the instruction to a static function_ref.

objc_method

```
sil-instruction ::= 'objc_method' sil-method-attributes?
                    sil-operand ',' sil-decl-ref ':' sil-type

%1 = objc_method %0 : $T, #T.method!foreign : $@convention(objc_method) U -> V
// %0 must be of a class type or class metatype $T
// #T.method must be a reference to an Objective-C method of T or
// of one of its superclasses
// %1 will be of type $U -> V
```

Performs Objective-C method dispatch using `objc_msgSend()`.

Objective-C method calls are never candidates for devirtualization.

super_method

```
sil-instruction ::= 'super_method' sil-method-attributes?
                    sil-operand ',' sil-decl-ref ':' sil-type

%1 = super_method %0 : $T, #Super.method : $@convention(thin) U -> V
// %0 must be of a non-root class type or class metatype $T
// #Super.method must be a reference to a native Swift method of T's
// superclass or of one of its ancestor classes
// %1 will be of type $@convention(thin) U -> V
```

Looks up a method in the superclass of a class or class metatype instance.

objc_super_method

```
sil-instruction ::= 'super_method' sil-method-attributes?
                    sil-operand ',' sil-decl-ref ':' sil-type

%1 = super_method %0 : $T, #Super.method!foreign : $@convention(thin) U -> V
// %0 must be of a non-root class type or class metatype $T
// #Super.method!foreign must be a reference to an ObjC method of T's
// superclass or of one of its ancestor classes
// %1 will be of type $@convention(thin) U -> V
```

This instruction performs an Objective-C message send using `objc_msgSuper()`.

witness_method

```
sil-instruction ::= 'witness_method' sil-method-attributes?
                    sil-type ',' sil-decl-ref ':' sil-type

%1 = witness_method $T, #Proto.method \
  : $@convention(witness_method) <Self: Proto> U -> V
// $T must be an archetype
// #Proto.method must be a reference to a method of one of the protocol
//   constraints on T
```

```
// <Self: Proto> U -> V must be the type of the referenced method,
//   generic on Self
// %1 will be of type $@convention(thin) <Self: Proto> U -> V
```

Looks up the implementation of a protocol method for a generic type variable constrained by that protocol. The result will be generic on the `Self` archetype of the original protocol and have the `witness_method` calling convention. If the referenced protocol is an `@objc` protocol, the resulting type has the `objc` calling convention.

Function Application

These instructions call functions or wrap them in partial application or specialization thunks.

apply

```
sil-instruction ::= 'apply' '[nothrow]'? sil-value
                     sil-apply-substitution-list?
                     '(' (sil-value (',' sil-value)*)? ')'
                     ':' sil-type

sil-apply-substitution-list ::= '<' sil-substitution
                                 (',' sil-substitution)* '>'
sil-substitution ::= type '=' type

%r = apply %0(%1, %2, ...) : $(A, B, ...) -> R
// Note that the type of the callee '%0' is specified *after* the arguments
// %0 must be of a concrete function type $(A, B, ...) -> R
// %1, %2, etc. must be of the argument types $A, $B, etc.
// %r will be of the return type $R

%r = apply %0<A, B>(%1, %2, ...) : $<T, U>(T, U, ...) -> R
// %0 must be of a polymorphic function type $<T, U>(T, U, ...) -> R
// %1, %2, etc. must be of the argument types after substitution $A, $B, etc.
// %r will be of the substituted return type $R'
```

Transfers control to function `%0`, passing it the given arguments. In the instruction syntax, the type of the callee is specified after the argument list; the types of the argument and of the defined value are derived from the function type of the callee. The input argument tuple type is destructured, and each element is passed as an individual argument. The `apply` instruction does no retaining or releasing of its arguments by itself; the calling convention's retain/release policy must be handled by separate explicit `retain` and `release` instructions. The return value will likewise not be implicitly retained or released.

The callee value must have function type. That function type may not have an error result, except the instruction has the `nothrow` attribute set. The `nothrow` attribute specifies that the callee has an error result but does not actually throw. For the regular case of calling a function with error result, use `try_apply`.

NB: If the callee value is of a thick function type, `apply` currently consumes the callee value at +1 strong retain count.

If the callee is generic, all of its generic parameters must be bound by the given substitution list. The arguments and return value is given with these generic substitutions applied.

begin_apply

```
sil-instruction ::= 'begin_apply' '[nothrow]'? sil-value
                     sil-apply-substitution-list?
                     '(' (sil-value (',' sil-value)*)? ')'
                     ':' sil-type

(%anyAddr, %float, %token) = begin_apply %0() : $@yield_once () -> (@yields @inout %Any, @yields Float)
// %anyAddr : $*Any
// %float : $Float
// %token is a token
```

Transfers control to coroutine `%0`, passing it the given arguments. The rules for the application generally follow the rules for `apply`, except:

- the callee value must have a `yield_once` coroutine type,
- control returns to this function not when the coroutine performs a `return`, but when it performs a `yield`, and
- the instruction results are derived from the yields of the coroutine instead of its normal results.

The final result of a `begin_apply` is a "token", a special value which can only be used as the operand of an `end_apply` or `abort_apply` instruction. Before this second instruction is executed, the coroutine is said to be "suspended", and the token represents a reference to its suspended activation record.

The other results of the instruction correspond to the yields in the coroutine type. In general, the rules of a yield are similar to the rules for a parameter, interpreted as if the coroutine caller (the one executing the `begin_apply`) were being "called" by the `yield`:

- If a yield has an indirect convention, the corresponding result will have an address type; otherwise it has an object type. For example, a result corresponding to an `@in Any` yield will have type `$Any`.
- The convention attributes are the same as the parameter convention attributes, interpreted as if the `yield` were the "call" and the `begin_apply` marked the entry to the "callee". For example, an `@in Any` yield transfers ownership of the `Any` value reference from the coroutine to the caller, which must destroy or move the value from that position before ending or aborting the coroutine.

A `begin_apply` must be uniquely either ended or aborted before exiting the function or looping to an earlier portion of the function.

When throwing coroutines are supported, there will need to be a `try_begin_apply` instruction.

abort_apply

```
sil-instruction ::= 'abort_apply' sil-value

abort_apply %token
```

Aborts the given coroutine activation, which is currently suspended at a `yield` instruction. Transfers control to the coroutine and takes the `unwind` path from the `yield`. Control is transferred back when the coroutine reaches an `unwind` instruction.

The operand must always be the token result of a `begin_apply` instruction, which is why it need not specify a type.

Throwing coroutines will not require a new instruction for aborting a coroutine; a coroutine is not allowed to throw when it is being aborted.

end_apply

```
sil-instruction ::= 'end_apply' sil-value

end_apply %token
```

Ends the given coroutine activation, which is currently suspended at a `yield` instruction. Transfers control to the coroutine and takes the `resume` path from the `yield`. Control is transferred back when the coroutine reaches a `return` instruction.

The operand must always be the token result of a `begin_apply` instruction, which is why it need not specify a type.

`end_apply` currently has no instruction results. If coroutines were allowed to have normal results, they would be producted by `end_apply`.

When throwing coroutines are supported, there will need to be a `try_end_apply` instruction.

partial_apply

```
sil-instruction ::= 'partial_apply' callee-ownership-attr? on-stack-attr? sil-value
                     sil-apply-substitution-list?
                     '(' (sil-value (',' sil-value)*)? ')'
                     ':' sil-type
callee-ownership-attr ::= '[callee_guaranteed]'
on-stack-attr ::= '[on_stack]'
```

```
%c = partial_apply %0(%1, %2, ...) : $(Z..., A, B, ...) -> R
// Note that the type of the callee '%0' is specified *after* the arguments
// %0 must be of a concrete function type $(Z..., A, B, ...) -> R
// %1, %2, etc. must be of the argument types $A, $B, etc.,
//   of the tail part of the argument tuple of %0
// %c will be of the partially-applied thick function type (Z...) -> R

%c = partial_apply %0<A, B>(%1, %2, ...) : $(Z..., T, U, ...) -> R
// %0 must be of a polymorphic function type $<T, U>(T, U, ...) -> R
// %1, %2, etc. must be of the argument types after substitution $A, $B, etc.
//   of the tail part of the argument tuple of %0
// %r will be of the substituted thick function type $(Z'...) -> R'
```

Creates a closure by partially applying the function `%0` to a partial sequence of its arguments. This instruction is used to implement closures.

A local function in Swift that captures context, such as `bar` in the following example:

```
func foo(_ x:Int) -> Int {
  func bar(_ y:Int) -> Int {
    return x + y
  }
  return bar(1)
}
```

lowers to an uncurried entry point and is curried in the enclosing function:

```
func @bar : $@convention(thin) (Int, @box Int, *Int) -> Int {
entry(%y : $Int, %x_box : $@box Int, %x_address : $*Int):
  // ... body of bar ...
}

func @foo : $@convention(thin) Int -> Int {
entry(%x : $Int):
  // Create a box for the 'x' variable
  %x_box = alloc_box $Int
  %x_addr = project_box %x_box : $@box Int
  store %x to %x_addr : $*Int

  // Create the bar closure
  %bar_uncurried = function_ref @bar : $(Int, Int) -> Int
  %bar = partial_apply %bar_uncurried(%x_box, %x_addr) \
    : $(Int, Builtin.NativeObject, *Int) -> Int

  // Apply it
  %1 = integer_literal $Int, 1
  %ret = apply %bar(%1) : $(Int) -> Int

  // Clean up
  release %bar : $(Int) -> Int
  return %ret : $Int
}
```

Ownership Semantics of Closure Context during Invocation: By default, an escaping `partial_apply` (`partial_apply` without `[on_stack]`) creates a closure whose invocation takes ownership of the context, meaning that a call implicitly releases the closure. If the `partial_apply` is marked with the flag `[callee_guaranteed]` the invocation instead uses a caller-guaranteed model, where the caller promises not to release the closure while the function is being called.

Captured Value Ownership Semantics: In the instruction syntax, the type of the callee is specified after the argument list; the types of the argument and of the defined value are derived from the function type of the callee. Even so, the ownership requirements of the partial apply are not the same as that of the callee function (and thus said signature). Instead:

1. If the `partial_apply` has a `@noescape` function type (`partial_apply [on_stack]`) the closure context is allocated on the stack and is initialized to contain the closed-over values without taking ownership of those values. The closed-over values are not retained and the lifetime of the closed-over values must be managed by other instruction independently of the `partial_apply`. The lifetime of the stack context of a `partial_apply [on_stack]` must be terminated with a `dealloc_stack`.
2. If the `partial_apply` has an escaping function type (not `[on_stack]`) then the closure context will be heap allocated with a retain count of 1. Any closed over parameters (except for `@inout` parameters) will be consumed by the partial_apply. This ensures that no matter when the `partial_apply` is called, the captured arguments are alive. When the closure context's reference count reaches zero, the contained values are destroyed. If the callee requires an owned parameter, then the implicit partial_apply forwarder created by IRGen will copy the underlying argument and pass it to the callee.
3. If an address argument has `@inout_aliasable` convention, the closure obtained from `partial_apply` will not own its underlying value. The `@inout_aliasable` parameter convention is used when a `@noescape` closure captures an `inout` argument.

NOTE: If the callee is generic, all of its generic parameters must be bound by the given substitution list. The arguments are given with these generic substitutions applied, and the resulting closure is of concrete function type with the given substitutions applied. The generic parameters themselves cannot be partially applied; all of them must be bound. The result is always a concrete function.

TODO: The instruction, when applied to a generic function, currently implicitly performs abstraction difference transformations enabled by the given substitutions, such as promoting address-only arguments and returns to register arguments. This should be fixed.

builtin

```
sil-instruction ::= 'builtin' string-literal
                      sil-apply-substitution-list?
                      '(' (sil-operand (',' sil-operand)*)? ')'
                      ':' sil-type

%1 = builtin "foo"(%1 : $T, %2 : $U) : $V
// "foo" must name a function in the Builtin module
```

Invokes functionality built into the backend code generator, such as LLVM- level instructions and intrinsics.

Metatypes

These instructions access metatypes, either statically by type name or dynamically by introspecting class or generic values.

metatype

```
sil-instruction ::= 'metatype' sil-type

%1 = metatype $T.Type
// %1 has type $T.Type
```

Creates a reference to the metatype object for type `T`.

value_metatype

```
sil-instruction ::= 'value_metatype' sil-type ',' sil-operand

%1 = value_metatype $T.Type, %0 : $T
// %0 must be a value or address of type $T
// %1 will be of type $T.Type
```

Obtains a reference to the dynamic metatype of the value `%0`.

existential_metatype

```
sil-instruction ::= 'existential_metatype' sil-type ',' sil-operand

%1 = existential_metatype $P.Type, %0 : $P
// %0 must be a value of class protocol or protocol composition
```

```
//   type $P, or an address of address-only protocol type $*P
// %1 will be a $P.Type value referencing the metatype of the
//   concrete value inside %0
```

Obtains the metatype of the concrete value referenced by the existential container referenced by `%0`.

objc_protocol

```
sil-instruction ::= 'objc_protocol' protocol-decl : sil-type

%0 = objc_protocol #ObjCProto : $Protocol
```

TODO Fill this in.

Aggregate Types

These instructions construct and project elements from structs, tuples, and class instances.

retain_value

```
sil-instruction ::= 'retain_value' sil-operand

retain_value %0 : $A
```

Retains a loadable value, which simply retains any references it holds.

For trivial types, this is a no-op. For reference types, this is equivalent to a `strong_retain`. For `@unowned` types, this is equivalent to an `unowned_retain`. In each of these cases, those are the preferred forms.

For aggregate types, especially enums, it is typically both easier and more efficient to reason about aggregate copies than it is to reason about copies of the subobjects.

retain_value_addr

```
sil-instruction ::= 'retain_value_addr' sil-operand

retain_value_addr %0 : $*A
```

Retains a loadable value inside given address, which simply retains any references it holds.

unmanaged_retain_value

```
sil-instruction ::= 'unmanaged_retain_value' sil-value

unmanaged_retain_value %0 : $A
```

This instruction has the same local semantics as `retain_value` but:

- Is valid in ownership qualified SIL.
- Is not intended to be statically paired at compile time by the compiler.

The intention is that this instruction is used to implement unmanaged constructs.

strong_copy_unmanaged_value

```
sil-instruction ::= 'strong_copy_unmanaged_value' sil-value

%1 = strong_copy_unmanaged_value %0 : $@sil_unmanaged A
// %1 will be a strong @owned $A.
```

This instruction has the same semantics as `copy_value` except that its input is a trivial `@sil_unmanaged` type that doesn't require ref counting. This is intended to be used semantically as a "conversion" like instruction from `unmanaged` to `strong` and thus should never be removed by the optimizer. Since the returned value is a strong owned value, this instruction semantically should be treated as performing a strong copy of the underlying value as if by the value's type lowering.

copy_value

```
sil-instruction ::= 'copy_value' sil-operand

%1 = copy_value %0 : $A
```

Performs a copy of a loadable value as if by the value's type lowering and returns the copy. The returned copy semantically is a value that is completely independent of the operand. In terms of specific types:

1. For trivial types, this is equivalent to just propagating through the trivial value.
2. For reference types, this is equivalent to performing a `strong_retain` operation and returning the reference.
3. For `@unowned` types, this is equivalent to performing an `unowned_retain` and returning the operand.
4. For aggregate types, this is equivalent to recursively performing a `copy_value` on its components, forming a new aggregate from the copied components, and then returning the new aggregate.

In ownership qualified functions, a `copy_value` produces a +1 value that must be consumed at most once along any path through the program.

explicit_copy_value

```
sil-instruction ::= 'explicit_copy_value' sil-operand

%1 = explicit_copy_value %0 : $A
```

Performs a copy of a loadable value as if by the value's type lowering and returns the copy. The returned copy semantically is a value that is completely independent of the operand. In terms of specific types:

1. For trivial types, this is equivalent to just propagating through the trivial value.
2. For reference types, this is equivalent to performing a `strong_retain` operation and returning the reference.
3. For `@unowned` types, this is equivalent to performing an `unowned_retain` and returning the operand.
4. For aggregate types, this is equivalent to recursively performing a `copy_value` on its components, forming a new aggregate from the copied components, and then returning the new aggregate.

In ownership qualified functions, a `explicit_copy_value` produces a +1 value that must be consumed at most once along any path through the program.

When move only variable checking is performed, `explicit_copy_value` is treated as an explicit copy asked for by the user that should not be rewritten and should be treated as a non-consuming use.

move_value

```
sil-instruction ::= 'move_value' '[lexical]'? sil-operand

%1 = move_value %0 : $@_moveOnly A
```

Performs a move of the operand, ending its lifetime. When ownership is enabled, it always takes in an *@owned T* and produces a new *@owned @_moveOnly T*.

1. For trivial types, this is equivalent to just propagating through the trivial value.
2. For reference types, this is equivalent to ending the lifetime of the operand, beginning a new lifetime for the result and setting the result to the value of the operand.
3. For aggregates, the operation is equivalent to performing a move_value on each of its fields recursively.

After ownership is lowered, we leave in the move_value to provide a place for IRGenSIL to know to store a potentially new variable (in case the move was associated with a let binding).

NOTE: This instruction is used in an experimental feature called 'move only values'. A move_value instruction is an instruction that introduces (or injects) a type *T* into the move only value space.

The `lexical` attribute specifies that the value corresponds to a local variable in the Swift source.

release_value

```
sil-instruction ::= 'release_value' sil-operand

release_value %0 : $A
```

Destroys a loadable value, by releasing any retainable pointers within it.

This is defined to be equivalent to storing the operand into a stack allocation and using 'destroy_addr' to destroy the object there.

For trivial types, this is a no-op. For reference types, this is equivalent to a `strong_release`. For `@unowned` types, this is equivalent to an `unowned_release`. In each of these cases, those are the preferred forms.

For aggregate types, especially enums, it is typically both easier and more efficient to reason about aggregate destroys than it is to reason about destroys of the subobjects.

release_value_addr

```
sil-instruction ::= 'release_value_addr' sil-operand

release_value_addr %0 : $*A
```

Destroys a loadable value inside given address, by releasing any retainable pointers within it.

unmanaged_release_value

```
sil-instruction ::= 'unmanaged_release_value' sil-value

unmanaged_release_value %0 : $A
```

This instruction has the same local semantics as `release_value` but:

- Is valid in ownership qualified SIL.
- Is not intended to be statically paired at compile time by the compiler.

The intention is that this instruction is used to implement unmanaged constructs.

destroy_value

```
sil-instruction ::= 'destroy_value' '[poison]'? sil-operand

destroy_value %0 : $A
```

Destroys a loadable value, by releasing any retainable pointers within it.

This is defined to be equivalent to storing the operand into a stack allocation and using 'destroy_addr' to destroy the object there.

For trivial types, this is a no-op. For reference types, this is equivalent to a `strong_release`. For `@unowned` types, this is equivalent to an `unowned_release`. In each of these cases, those are the preferred forms.

For aggregate types, especially enums, it is typically both easier and more efficient to reason about aggregate destroys than it is to reason about destroys of the subobjects.

autorelease_value

```
sil-instruction ::= 'autorelease_value' sil-operand

autorelease_value %0 : $A
```

TODO Complete this section.

tuple

```
sil-instruction ::= 'tuple' sil-tuple-elements
sil-tuple-elements ::= '(' (sil-operand (',' sil-operand)*)? ')'
sil-tuple-elements ::= sil-type '(' (sil-value (',' sil-value)*)? ')'

%1 = tuple (%a : $A, %b : $B, ...)
// $A, $B, etc. must be loadable non-address types
// %1 will be of the "simple" tuple type $(A, B, ...)

%1 = tuple $(a:A, b:B, ...) (%a, %b, ...)
// (a:A, b:B, ...) must be a loadable tuple type
// %1 will be of the type $(a:A, b:B, ...)
```

Creates a loadable tuple value by aggregating multiple loadable values.

If the destination type is a "simple" tuple type, that is, it has no keyword argument labels or variadic arguments, then the first notation can be used, which interleaves the element values and types. If keyword names or variadic fields are specified, then the second notation must be used, which spells out the tuple type before the fields.

tuple_extract

```
sil-instruction ::= 'tuple_extract' sil-operand ',' int-literal

%1 = tuple_extract %0 : $(T...), 123
// %0 must be of a loadable tuple type $(T...)
// %1 will be of the type of the selected element of %0
```

Extracts an element from a loadable tuple value.

tuple_element_addr

```
sil-instruction ::= 'tuple_element_addr' sil-operand ',' int-literal

%1 = tuple_element_addr %0 : $*(T...), 123
// %0 must of a $*(T...) address-of-tuple type
// %1 will be of address type $*U where U is the type of the 123rd
//   element of T
```

Given the address of a tuple in memory, derives the address of an element within that value.

destructure_tuple

```
sil-instruction ::= 'destructure_tuple' sil-operand

(%elt1, ..., %eltn) = destructure_tuple %0 : $(Elt1Ty, ..., EltNTy)
// %0 must be a tuple of type $(Elt1Ty, ..., EltNTy)
// %eltN must have the type $EltNTy
```

Given a tuple value, split the value into its constituent elements.

struct

```
sil-instruction ::= 'struct' sil-type '(' (sil-operand (',' sil-operand)*)? ')'

%1 = struct $S (%a : $A, %b : $B, ...)
// $S must be a loadable struct type
// $A, $B, ... must be the types of the physical 'var' fields of $S in order
// %1 will be of type $S
```

Creates a value of a loadable struct type by aggregating multiple loadable values.

struct_extract

```
sil-instruction ::= 'struct_extract' sil-operand ',' sil-decl-ref

%1 = struct_extract %0 : $S, #S.field
// %0 must be of a loadable struct type $S
// #S.field must be a physical 'var' field of $S
// %1 will be of the type of the selected field of %0
```

Extracts a physical field from a loadable struct value.

struct_element_addr

```
sil-instruction ::= 'struct_element_addr' sil-operand ',' sil-decl-ref

%1 = struct_element_addr %0 : $*S, #S.field
// %0 must be of a struct type $S
// #S.field must be a physical 'var' field of $S
// %1 will be the address of the selected field of %0
```

Given the address of a struct value in memory, derives the address of a physical field within the value.

destructure_struct

```
sil-instruction ::= 'destructure_struct' sil-operand

(%elt1, ..., %eltn) = destructure_struct %0 : $S
// %0 must be a struct of type $S
// %eltN must have the same type as the Nth field of $S
```

Given a struct, split the struct into its constituent fields.

object

```
sil-instruction ::= 'object' sil-type '(' (sil-operand (',' sil-operand)*)? ')'

object $T (%a : $A, %b : $B, ...)
// $T must be a non-generic or bound generic reference type
// The first operands must match the stored properties of T
// Optionally there may be more elements, which are tail-allocated to T
```

Constructs a statically initialized object. This instruction can only appear as final instruction in a global variable static initializer list.

ref_element_addr

```
sil-instruction ::= 'ref_element_addr' '[immutable]'? sil-operand ',' sil-decl-ref

%1 = ref_element_addr %0 : $C, #C.field
// %0 must be a value of class type $C
// #C.field must be a non-static physical field of $C
// %1 will be of type $*U where U is the type of the selected field
//   of C
```

Given an instance of a class, derives the address of a physical instance variable inside the instance. It is undefined behavior if the class value is null.

The `immutable` attribute specifies that all loads of the same instance variable from the same class reference operand are guaranteed to yield the same value.

ref_tail_addr

```
sil-instruction ::= 'ref_tail_addr' '[immutable]'? sil-operand ',' sil-type

%1 = ref_tail_addr %0 : $C, $E
// %0 must be a value of class type $C with tail-allocated elements $E
// %1 will be of type $*E
```

Given an instance of a class, which is created with tail-allocated array(s), derives the address of the first element of the first tail-allocated array. This instruction is used to project the first tail-allocated element from an object which is created by an `alloc_ref` with `tail_elems`. It is undefined behavior if the class instance does not have tail-allocated arrays or if the element-types do not match.

The `immutable` attribute specifies that all loads of the same instance variable from the same class reference operand are guaranteed to yield the same value.

Enums

These instructions construct and manipulate values of enum type. Loadable enum values are created with the enum instruction. Address-only enums require two-step initialization. First, if the case requires data, that data is stored into the enum at the address projected by init_enum_data_addr. This step is skipped for cases without data. Finally, the tag for the enum is injected with an inject_enum_addr instruction:

```
enum AddressOnlyEnum {
  case HasData(AddressOnlyType)
  case NoData
}

sil @init_with_data : $(AddressOnlyType) -> AddressOnlyEnum {
entry(%0 : $*AddressOnlyEnum, %1 : $*AddressOnlyType):
  // Store the data argument for the case.
  %2 = init_enum_data_addr %0 : $*AddressOnlyEnum, #AddressOnlyEnum.HasData!enumelt
  copy_addr [take] %2 to [initialization] %1 : $*AddressOnlyType
  // Inject the tag.
  inject_enum_addr %0 : $*AddressOnlyEnum, #AddressOnlyEnum.HasData!enumelt
  return
}

sil @init_without_data : $() -> AddressOnlyEnum {
  // No data. We only need to inject the tag.
  inject_enum_addr %0 : $*AddressOnlyEnum, #AddressOnlyEnum.NoData!enumelt
  return
}
```

Accessing the value of a loadable enum is inseparable from dispatching on its discriminator and is done with the switch_enum terminator:

```
enum Foo { case A(Int), B(String) }

sil @switch_foo : $(Foo) -> () {
entry(%foo : $Foo):
  switch_enum %foo : $Foo, case #Foo.A!enumelt: a_dest, case #Foo.B!enumelt: b_dest

a_dest(%a : $Int):
  /* use %a */

b_dest(%b : $String):
  /* use %b */
}
```

An address-only enum can be tested by branching on it using the switch_enum_addr terminator. Its value can then be taken by destructively projecting the enum value with unchecked_take_enum_data_addr:

```
enum Foo<T> { case A(T), B(String) }

sil @switch_foo : $<T> (Foo<T>) -> () {
entry(%foo : $*Foo<T>):
  switch_enum_addr %foo : $*Foo<T>, case #Foo.A!enumelt: a_dest, \
    case #Foo.B!enumelt: b_dest
```

```
a_dest:
  %a = unchecked_take_enum_data_addr %foo : $*Foo<T>, #Foo.A!enumelt
  /* use %a */

b_dest:
  %b = unchecked_take_enum_data_addr %foo : $*Foo<T>, #Foo.B!enumelt
  /* use %b */
}
```

Both switch_enum and switch_enum_addr must include a `default` case unless the enum can be exhaustively switched in the current function, i.e. when the compiler can be sure that it knows all possible present and future values of the enum in question. This is generally true for enums defined in Swift, but there are two exceptions: *non-frozen enums* declared in libraries compiled with the `-enable-library-evolution` flag, which may grow new cases in the future in an ABI-compatible way; and enums marked with the `objc` attribute, for which other bit patterns are permitted for compatibility with C. All enums imported from C are treated as "non-exhaustive" for the same reason, regardless of the presence or value of the `enum_extensibility` Clang attribute.

(See SE-0192 for more information about non-frozen enums.)

enum

```
sil-instruction ::= 'enum' sil-type ',' sil-decl-ref (',' sil-operand)?

%1 = enum $U, #U.EmptyCase!enumelt
%1 = enum $U, #U.DataCase!enumelt, %0 : $T
// $U must be an enum type
// #U.DataCase or #U.EmptyCase must be a case of enum $U
// If #U.Case has a data type $T, %0 must be a value of type $T
// If #U.Case has no data type, the operand must be omitted
// %1 will be of type $U
```

Creates a loadable enum value in the given `case`. If the `case` has a data type, the enum value will contain the operand value.

unchecked_enum_data

```
sil-instruction ::= 'unchecked_enum_data' sil-operand ',' sil-decl-ref

%1 = unchecked_enum_data %0 : $U, #U.DataCase!enumelt
// $U must be an enum type
// #U.DataCase must be a case of enum $U with data
// %1 will be of object type $T for the data type of case U.DataCase
```

Unsafely extracts the payload data for an enum `case` from an enum value. It is undefined behavior if the enum does not contain a value of the given case.

init_enum_data_addr

```
sil-instruction ::= 'init_enum_data_addr' sil-operand ',' sil-decl-ref

%1 = init_enum_data_addr %0 : $*U, #U.DataCase!enumelt
// $U must be an enum type
// #U.DataCase must be a case of enum $U with data
// %1 will be of address type $*T for the data type of case U.DataCase
```

Projects the address of the data for an enum `case` inside an enum. This does not modify the enum or check its value. It is intended to be used as part of the initialization sequence for an address-only enum. Storing to the `init_enum_data_addr` for a case followed by `inject_enum_addr` with that same case is guaranteed to result in a fully-initialized enum value of that case being stored. Loading from the `init_enum_data_addr` of an initialized enum value or injecting a mismatched case tag is undefined behavior.

The address is invalidated as soon as the operand enum is fully initialized by an `inject_enum_addr`.

inject_enum_addr

```
sil-instruction ::= 'inject_enum_addr' sil-operand ',' sil-decl-ref

inject_enum_addr %0 : $*U, #U.Case!enumelt
// $U must be an enum type
// #U.Case must be a case of enum $U
// %0 will be overlaid with the tag for #U.Case
```

Initializes the enum value referenced by the given address by overlaying the tag for the given case. If the case has no data, this instruction is sufficient to initialize the enum value. If the case has data, the data must be stored into the enum at the `init_enum_data_addr` address for the case *before* `inject_enum_addr` is applied. It is undefined behavior if `inject_enum_addr` is applied for a case with data to an uninitialized enum, or if `inject_enum_addr` is applied for a case with data when data for a mismatched case has been stored to the enum.

unchecked_take_enum_data_addr

```
sil-instruction ::= 'unchecked_take_enum_data_addr' sil-operand ',' sil-decl-ref

%1 = unchecked_take_enum_data_addr %0 : $*U, #U.DataCase!enumelt
// $U must be an enum type
// #U.DataCase must be a case of enum $U with data
// %1 will be of address type $*T for the data type of case U.DataCase
```

Invalidates an enum value, and takes the address of the payload for the given enum `case` in-place in memory. The referenced enum value is no longer valid, but the payload value referenced by the result address is valid and must be destroyed. It is undefined behavior if the referenced enum does not contain a value of the given `case`. The result shares memory with the original enum value; the enum memory cannot be reinitialized as an enum until the payload has also been invalidated.

(1.0 only)

For the first payloaded case of an enum, `unchecked_take_enum_data_addr` is guaranteed to have no side effects; the enum value will not be invalidated.

select_enum

```
sil-instruction ::= 'select_enum' sil-operand sil-select-case*
                    (',' 'default' sil-value)?
                    ':' sil-type

%n = select_enum %0 : $U,      \
  case #U.Case1!enumelt: %1,           \
  case #U.Case2!enumelt: %2, /* ... */ \
  default %3 : $T

// $U must be an enum type
// #U.Case1, Case2, etc. must be cases of enum $U
// %1, %2, %3, etc. must have type $T
// %n has type $T
```

Selects one of the "case" or "default" operands based on the case of an enum value. This is equivalent to a trivial switch_enum branch sequence:

```
entry:
  switch_enum %0 : $U,           \
    case #U.Case1!enumelt: bb1,          \
    case #U.Case2!enumelt: bb2, /* ... */ \
    default bb_default
bb1:
  br cont(%1 : $T) // value for #U.Case1
bb2:
  br cont(%2 : $T) // value for #U.Case2
bb_default:
```

```
  br cont(%3 : $T) // value for default
cont(%n : $T):
  // use argument %n
```

but turns the control flow dependency into a data flow dependency. For address-only enums, select_enum_addr offers the same functionality for an indirectly referenced enum value in memory.

Like switch_enum, select_enum must have a `default` case unless the enum can be exhaustively switched in the current function.

select_enum_addr

```
sil-instruction ::= 'select_enum_addr' sil-operand sil-select-case*
                    (',' 'default' sil-value)?
                    ':' sil-type

%n = select_enum_addr %0 : $*U,      \
  case #U.Case1!enumelt: %1,         \
  case #U.Case2!enumelt: %2, /* ... */ \
  default %3 : $T

// %0 must be the address of an enum type $*U
// #U.Case1, Case2, etc. must be cases of enum $U
// %1, %2, %3, etc. must have type $T
// %n has type $T
```

Selects one of the "case" or "default" operands based on the case of the referenced enum value. This is the address-only counterpart to select_enum.

Like switch_enum_addr, select_enum_addr must have a `default` case unless the enum can be exhaustively switched in the current function.

Protocol and Protocol Composition Types

These instructions create and manipulate values of protocol and protocol composition type. From SIL's perspective, protocol and protocol composition types consist of an *existential container*, which is a generic container for a value of unknown runtime type, referred to as an "existential type" in type theory. The existential container consists of a reference to the *witness table(s)* for the protocol(s) referred to by the protocol type and a reference to the underlying *concrete value*, which may be either stored in-line inside the existential container for small values or allocated separately into a buffer owned and managed by the existential container for larger values.

Depending on the constraints applied to an existential type, an existential container may use one of several representations:

- **Opaque existential containers**: If none of the protocols in a protocol type are class protocols, then the existential container for that type is address-only and referred to in the implementation as an *opaque existential container*. The value semantics of the existential container propagate to the contained concrete value. Applying copy_addr to an opaque existential container copies the contained concrete value, deallocating or reallocating the destination container's owned buffer if necessary. Applying destroy_addr to an opaque existential container destroys the concrete value and deallocates any buffers owned by the existential container. The following instructions manipulate opaque existential containers:
  - init_existential_addr
  - open_existential_addr
  - deinit_existential_addr
- **Opaque existential containers loadable types**: In the SIL Opaque Values mode of operation, we take an opaque value as-is. Said value might be replaced with one of the _addr instructions above before IR generation. The following instructions manipulate "loadable" opaque existential containers:
  - init_existential_value
  - open_existential_value
  - deinit_existential_value
- **Class existential containers**: If a protocol type is constrained by one or more class protocols, then the existential container for that type is loadable and referred to in the implementation as a *class existential container*. Class existential containers have reference semantics and can be `retain`-ed and `release`-d. The following instructions manipulate class existential containers:
  - init_existential_ref
  - open_existential_ref
- **Metatype existential containers**: Existential metatypes use a container consisting of the type metadata for the conforming type along with the protocol conformances. Metatype existential containers are trivial types. The following instructions manipulate metatype existential containers:
  - init_existential_metatype
  - open_existential_metatype
- **Boxed existential containers**: The standard library `Error` protocol uses a size-optimized reference-counted container, which indirectly stores the conforming value. Boxed existential containers can be `retain`-ed and `release`-d. The following instructions manipulate boxed existential containers:
  - alloc_existential_box
  - project_existential_box
  - open_existential_box
  - open_existential_box_value
  - dealloc_existential_box

Some existential types may additionally support specialized representations when they contain certain known concrete types. For example, when Objective-C interop is available, the `Error` protocol existential supports a class existential container representation for `NSError` objects, so it can be initialized from one using init_existential_ref instead of the more expensive alloc_existential_box:

```
bb(%nserror: $NSError):
  // The slow general way to form an Error, allocating a box and
  // storing to its value buffer:
  %error1 = alloc_existential_box $Error, $NSError
  %addr = project_existential_box $NSError in %error1 : $Error
  strong_retain %nserror: $NSError
  store %nserror to %addr : $NSError

  // The fast path supported for NSError:
  strong_retain %nserror: $NSError
  %error2 = init_existential_ref %nserror: $NSError, $Error
```

init_existential_addr

```
sil-instruction ::= 'init_existential_addr' sil-operand ',' sil-type

%1 = init_existential_addr %0 : $*P, $T
// %0 must be of a $*P address type for non-class protocol or protocol
//   composition type P
// $T must be an AST type that fulfills protocol(s) P
// %1 will be of type $*T', where T' is the maximally abstract lowering
//    of type T
```

Partially initializes the memory referenced by `%0` with an existential container prepared to contain a value of type `$T`. The result of the instruction is an address referencing the storage for the contained value, which remains uninitialized. The contained value must be `store`-d or `copy_addr`-ed to in order for the existential value to be fully initialized. If the existential container needs to be destroyed while the contained value is uninitialized, deinit_existential_addr must be used to do so. A fully initialized existential container can be destroyed with destroy_addr as usual. It is undefined behavior to destroy_addr a partially-initialized existential container.

init_existential_value

```
sil-instruction ::= 'init_existential_value' sil-operand ',' sil-type ','
                                             sil-type

%1 = init_existential_value %0 : $L, $C, $P
// %0 must be of loadable type $L, lowered from AST type $C, conforming to
//    protocol(s) $P
```

```
// %1 will be of type $P
```

Loadable version of the above: Inits-up the existential container prepared to contain a value of type `$P`.

deinit_existential_addr

```
sil-instruction ::= 'deinit_existential_addr' sil-operand

deinit_existential_addr %0 : $*P
// %0 must be of a $*P address type for non-class protocol or protocol
// composition type P
```

Undoes the partial initialization performed by init_existential_addr. deinit_existential_addr is only valid for existential containers that have been partially initialized by init_existential_addr but haven't had their contained value initialized. A fully initialized existential must be destroyed with destroy_addr.

deinit_existential_value

```
sil-instruction ::= 'deinit_existential_value' sil-operand

deinit_existential_value %0 : $P
// %0 must be of a $P opaque type for non-class protocol or protocol
// composition type P
```

Undoes the partial initialization performed by init_existential_value. deinit_existential_value is only valid for existential containers that have been partially initialized by init_existential_value but haven't had their contained value initialized. A fully initialized existential must be destroyed with destroy_value.

open_existential_addr

```
sil-instruction ::= 'open_existential_addr' sil-allowed-access sil-operand 'to' sil-type
sil-allowed-access ::= 'immutable_access'
sil-allowed-access ::= 'mutable_access'

%1 = open_existential_addr immutable_access %0 : $*P to $*@opened P
// %0 must be of a $*P type for non-class protocol or protocol composition
//   type P
// $*@opened P must be a unique archetype that refers to an opened
// existential type P.
// %1 will be of type $*@opened P
```

Obtains the address of the concrete value inside the existential container referenced by `%0`. The protocol conformances associated with this existential container are associated directly with the archetype `$*@opened P`. This pointer can be used with any operation on archetypes, such as `witness_method` assuming this operation obeys the access constraint: The returned address can either allow `mutable_access` or `immutable_access`. Users of the returned address may only consume (e.g `destroy_addr` or `copy_addr [take]`) or mutate the value at the address if they have `mutable_access`.

open_existential_value

```
sil-instruction ::= 'open_existential_value' sil-operand 'to' sil-type

%1 = open_existential_value %0 : $P to $@opened P
// %0 must be of a $P type for non-class protocol or protocol composition
//   type P
// $@opened P must be a unique archetype that refers to an opened
// existential type P.
// %1 will be of type $@opened P
```

Loadable version of the above: Opens-up the existential container associated with `%0`. The protocol conformances associated with this existential container are associated directly with the archetype `$@opened P`.

init_existential_ref

```
sil-instruction ::= 'init_existential_ref' sil-operand ':' sil-type ','
                                           sil-type

%1 = init_existential_ref %0 : $C' : $C, $P
// %0 must be of class type $C', lowered from AST type $C, conforming to
//    protocol(s) $P
// $P must be a class protocol or protocol composition type
// %1 will be of type $P
```

Creates a class existential container of type `$P` containing a reference to the class instance `%0`.

open_existential_ref

```
sil-instruction ::= 'open_existential_ref' sil-operand 'to' sil-type

%1 = open_existential_ref %0 : $P to $@opened P
// %0 must be of a $P type for a class protocol or protocol composition
// $@opened P must be a unique archetype that refers to an opened
//   existential type P
// %1 will be of type $@opened P
```

Extracts the class instance reference from a class existential container. The protocol conformances associated with this existential container are associated directly with the archetype `@opened P`. This pointer can be used with any operation on archetypes, such as witness_method. When the operand is of metatype type, the result will be the metatype of the opened archetype.

init_existential_metatype

```
sil-instruction ::= 'init_existential_metatype' sil-operand ',' sil-type

%1 = init_existential_metatype $0 : $@<rep> T.Type, $@<rep> P.Type
// %0 must be of a metatype type $@<rep> T.Type where T: P
// %@<rep> P.Type must be the existential metatype of a protocol or protocol
//    composition, with the same metatype representation <rep>
// %1 will be of type $@<rep> P.Type
```

Creates a metatype existential container of type `$P.Type` containing the conforming metatype of `$T`.

open_existential_metatype

```
sil-instruction ::= 'open_existential_metatype' sil-operand 'to' sil-type

%1 = open_existential_metatype %0 : $@<rep> P.Type to $@<rep> (@opened P).Type
// %0 must be of a $P.Type existential metatype for a protocol or protocol
//    composition
// $@<rep> (@opened P).Type must be the metatype of a unique archetype that
//   refers to an opened existential type P, with the same metatype
//   representation <rep>
// %1 will be of type $@<rep> (@opened P).Type
```

Extracts the metatype from an existential metatype. The protocol conformances associated with this existential container are associated directly with the archetype `@opened P`.

alloc_existential_box

```
sil-instruction ::= 'alloc_existential_box' sil-type ',' sil-type

%1 = alloc_existential_box $P, $T
// $P must be a protocol or protocol composition type with boxed
//   representation
// $T must be an AST type that conforms to P
```

```
// %1 will be of type $P
```

Allocates a boxed existential container of type `$P` with space to hold a value of type `$T'`. The box is not fully initialized until a valid value has been stored into the box. If the box must be deallocated before it is fully initialized, dealloc_existential_box must be used. A fully initialized box can be `retain`-ed and `release`-d like any reference-counted type. The project_existential_box instruction is used to retrieve the address of the value inside the container.

project_existential_box

```
sil-instruction ::= 'project_existential_box' sil-type 'in' sil-operand

%1 = project_existential_box $T in %0 : $P
// %0 must be a value of boxed protocol or protocol composition type $P
// $T must be the most abstracted lowering of the AST type for which the box
// was allocated
// %1 will be of type $*T
```

Projects the address of the value inside a boxed existential container. The address is dependent on the lifetime of the owner reference `%0`. It is undefined behavior if the concrete type `$T` is not the same type for which the box was allocated with alloc_existential_box.

open_existential_box

```
sil-instruction ::= 'open_existential_box' sil-operand 'to' sil-type

%1 = open_existential_box %0 : $P to $*@opened P
// %0 must be a value of boxed protocol or protocol composition type $P
// %@opened P must be the address type of a unique archetype that refers to
///   an opened existential type P
// %1 will be of type $*@opened P
```

Projects the address of the value inside a boxed existential container, and uses the enclosed type and protocol conformance metadata to bind the opened archetype `$@opened P`. The result address is dependent on both the owning box and the enclosing function; in order to "open" a boxed existential that has directly adopted a class reference, temporary scratch space may need to have been allocated.

open_existential_box_value

```
sil-instruction ::= 'open_existential_box_value' sil-operand 'to' sil-type

%1 = open_existential_box_value %0 : $P to $@opened P
// %0 must be a value of boxed protocol or protocol composition type $P
// %@opened P must be a unique archetype that refers to an opened
//   existential type P
// %1 will be of type $@opened P
```

Projects the value inside a boxed existential container, and uses the enclosed type and protocol conformance metadata to bind the opened archetype `$@opened P`.

dealloc_existential_box

```
sil-instruction ::= 'dealloc_existential_box' sil-operand, sil-type

dealloc_existential_box %0 : $P, $T
// %0 must be an uninitialized box of boxed existential container type $P
// $T must be the AST type for which the box was allocated
```

Deallocates a boxed existential container. The value inside the existential buffer is not destroyed; either the box must be uninitialized, or the value must have been projected out and destroyed beforehand. It is undefined behavior if the concrete type `$T` is not the same type for which the box was allocated with alloc_existential_box.

Blocks

project_block_storage

```
sil-instruction ::= 'project_block_storage' sil-operand ':' sil-type
```

init_block_storage_header

TODO Fill this in. The printing of this instruction looks incomplete on trunk currently.

Unchecked Conversions

These instructions implement type conversions which are not checked. These are either user-level conversions that are always safe and do not need to be checked, or implementation detail conversions that are unchecked for performance or flexibility.

upcast

```
sil-instruction ::= 'upcast' sil-operand 'to' sil-type

%1 = upcast %0 : $D to $B
// $D and $B must be class types or metatypes, with B a superclass of D
// %1 will have type $B
```

Represents a conversion from a derived class instance or metatype to a superclass, or from a base-class-constrained archetype to its base class.

address_to_pointer

```
sil-instruction ::= 'address_to_pointer' sil-operand 'to' sil-type

%1 = address_to_pointer %0 : $*T to $Builtin.RawPointer
// %0 must be of an address type $*T
// %1 will be of type Builtin.RawPointer
```

Creates a `Builtin.RawPointer` value corresponding to the address `%0`. Converting the result pointer back to an address of the same type will give an address equivalent to `%0`. It is undefined behavior to cast the `RawPointer` to any address type other than its original address type or any layout compatible types.

pointer_to_address

```
sil-instruction ::= 'pointer_to_address' sil-operand 'to' ('[' 'strict' ']')? ('[' 'invariant' ']')? ('[' 'alignment' '=' alignment ']')?
alignment ::= [0-9]+

%1 = pointer_to_address %0 : $Builtin.RawPointer to [strict] $*T
// %1 will be of type $*T
```

Creates an address value corresponding to the `Builtin.RawPointer` value `%0`. Converting a `RawPointer` back to an address of the same type as its originating address_to_pointer instruction gives back an equivalent address. It is undefined behavior to cast the `RawPointer` back to any type other than its original address type or layout compatible types. It is also undefined behavior to cast a `RawPointer` from a heap object to any address type.

The `strict` flag indicates whether the returned address adheres to strict aliasing. If true, then the type of each memory access dependent on this address must be consistent with the memory's bound type. A memory access from an address that is not strict cannot have its address substituted with a strict address, even if other nearby memory accesses at the same location are strict.

The `invariant` flag is set if loading from the returned address always produces the same value.

The `alignment` integer value specifies the byte alignment of the address. `alignment=0` is the default, indicating the natural alignment of `T`.

unchecked_ref_cast

```
sil-instruction ::= 'unchecked_ref_cast' sil-operand 'to' sil-type

%1 = unchecked_ref_cast %0 : $A to $B
// %0 must be an object of type $A
// $A must be a type with retainable pointer representation
// %1 will be of type $B
// $B must be a type with retainable pointer representation
```

Converts a heap object reference to another heap object reference type. This conversion is unchecked, and it is undefined behavior if the destination type is not a valid type for the heap object. The heap object reference on either side of the cast may be a class existential, and may be wrapped in one level of Optional.

unchecked_ref_cast_addr

```
sil-instruction ::= 'unchecked_ref_cast_addr'
                    sil-type 'in' sil-operand 'to'
                    sil-type 'in' sil-operand

unchecked_ref_cast_addr $A in %0 : $*A to $B in %1 : $*B
// %0 must be the address of an object of type $A
// $A must be a type with retainable pointer representation
// %1 must be the address of storage for an object of type $B
// $B must be a retainable pointer representation
```

Loads a heap object reference from an address and stores it at the address of another uninitialized heap object reference. The loaded reference is always taken, and the stored reference is initialized. This conversion is unchecked, and it is undefined behavior if the destination type is not a valid type for the heap object. The heap object reference on either side of the cast may be a class existential, and may be wrapped in one level of Optional.

unchecked_addr_cast

```
sil-instruction ::= 'unchecked_addr_cast' sil-operand 'to' sil-type

%1 = unchecked_addr_cast %0 : $*A to $*B
// %0 must be an address
// %1 will be of type $*B
```

Converts an address to a different address type. Using the resulting address is undefined unless `B` is layout compatible with `A`. The layout of `B` may be smaller than that of `A` as long as the lower order bytes have identical layout.

unchecked_trivial_bit_cast

```
sil-instruction ::= 'unchecked_trivial_bit_cast' sil-operand 'to' sil-type

%1 = unchecked_trivial_bit_cast %0 : $Builtin.NativeObject to $Builtin.Word
// %0 must be an object.
// %1 must be an object with trivial type.
```

Bitcasts an object of type `A` to be of same sized or smaller type `B` with the constraint that `B` must be trivial. This can be used for bitcasting among trivial types, but more importantly is a one way bitcast from non-trivial types to trivial types.

unchecked_bitwise_cast

```
sil-instruction ::= 'unchecked_bitwise_cast' sil-operand 'to' sil-type

%1 = unchecked_bitwise_cast %0 : $A to $B
```

Bitwise copies an object of type `A` into a new object of type `B` of the same size or smaller.

unchecked_value_cast

```
sil-instruction ::= 'unchecked_value_cast' sil-operand 'to' sil-type

%1 = unchecked_value_cast %0 : $A to $B
```

Bitwise copies an object of type `A` into a new layout-compatible object of type `B` of the same size.

This instruction is assumed to forward a fixed ownership (set upon its construction) and lowers to 'unchecked_bitwise_cast' in non-ossa code. This causes the cast to lose its guarantee of layout-compatibility.

unchecked_ownership_conversion

```
sil-instruction ::= 'unchecked_ownership_conversion' sil-operand ',' sil-value-ownership-kind 'to' sil-value-ownership-kind

%1 = unchecked_ownership_conversion %0 : $A, @guaranteed to @owned
```

Converts its operand to an identical value of the same type but with different ownership without performing any semantic operations normally required by for ownership conversion.

This is used in Objective-C compatible destructors to convert a guaranteed parameter to an owned parameter without performing a semantic copy.

The resulting value must meet the usual ownership requirements; for example, a trivial type must have '.none' ownership.

ref_to_raw_pointer

```
sil-instruction ::= 'ref_to_raw_pointer' sil-operand 'to' sil-type

%1 = ref_to_raw_pointer %0 : $C to $Builtin.RawPointer
// $C must be a class type, or Builtin.NativeObject, or AnyObject
// %1 will be of type $Builtin.RawPointer
```

Converts a heap object reference to a `Builtin.RawPointer`. The `RawPointer` result can be cast back to the originating class type but does not have ownership semantics. It is undefined behavior to cast a `RawPointer` from a heap object reference to an address using pointer_to_address.

raw_pointer_to_ref

```
sil-instruction ::= 'raw_pointer_to_ref' sil-operand 'to' sil-type

%1 = raw_pointer_to_ref %0 : $Builtin.RawPointer to $C
// $C must be a class type, or Builtin.NativeObject, or AnyObject
// %1 will be of type $C
```

Converts a `Builtin.RawPointer` back to a heap object reference. Casting a heap object reference to `Builtin.RawPointer` back to the same type gives an equivalent heap object reference (though the raw pointer has no ownership semantics for the object on its own). It is undefined behavior to cast a `RawPointer` to a type unrelated to the dynamic type of the heap object. It is also undefined behavior to cast a `RawPointer` from an address to any heap object type.

ref_to_unowned

```
sil-instruction ::= 'ref_to_unowned' sil-operand

%1 = unowned_to_ref %0 : T
// $T must be a reference type
// %1 will have type $@unowned T
```

Adds the `@unowned` qualifier to the type of a reference to a heap object. No runtime effect.

unowned_to_ref

```
sil-instruction ::= 'unowned_to_ref' sil-operand
```

```
%1 = unowned_to_ref %0 : $@unowned T
// $T must be a reference type
// %1 will have type $T
```

Strips the `@unowned` qualifier off the type of a reference to a heap object. No runtime effect.

ref_to_unmanaged

TODO

unmanaged_to_ref

TODO

convert_function

```
sil-instruction ::= 'convert_function' sil-operand 'to'
                    ('[' 'without_actually_escaping' ']')?
                    sil-type

%1 = convert_function %0 : $T -> U to $T' -> U'
// %0 must be of a function type $T -> U ABI-compatible with $T' -> U'
//   (see below)
// %1 will be of type $T' -> U'
```

Performs a conversion of the function `%0` to type `T`, which must be ABI- compatible with the type of `%0`. Function types are ABI-compatible if their input and result types are tuple types that, after destructuring, differ only in the following ways:

- Corresponding tuple elements may add, remove, or change keyword names. `(a:Int, b:Float, UnicodeScalar) -> ()` and `(x:Int, Float, z:UnicodeScalar) -> ()` are ABI compatible.
- A class tuple element of the destination type may be a superclass or subclass of the source type's corresponding tuple element.

The function types may also differ in attributes, except that the `convention` attribute cannot be changed and the `@noescape` attribute must not change for functions with context.

A `convert_function` cannot be used to change a thick type's `@noescape` attribute (`@noescape` function types with context are not ABI compatible with escaping function types with context) -- however, thin function types with and without `@noescape` are ABI compatible because they have no context. To convert from an escaping to a `@noescape` thick function type use `convert_escape_to_noescape`.

With the `without_actually_escaping` attribute, the `convert_function` may be used to convert a non-escaping closure into an escaping function type. This attribute must be present whenever the closure operand has an unboxed capture (via @inout_aliasable) *and* the resulting function type is escaping. (This only happens as a result of withoutActuallyEscaping()). If the attribute is present then the resulting function type must be escaping, but the operand's function type may or may not be @noescape. Note that a non-escaping closure may have unboxed captured even though its SIL function type is "escaping".

convert_escape_to_noescape

```
sil-instruction ::= 'convert_escape_to_noescape' sil-operand 'to' sil-type
%1 = convert_escape_to_noescape %0 : $T -> U to $@noescape T' -> U'
// %0 must be of a function type $T -> U ABI-compatible with $T' -> U'
//   (see convert_function)
// %1 will be of the trivial type $@noescape T -> U
```

Converts an escaping (non-trivial) function type to a `@noescape` trivial function type. Something must guarantee the lifetime of the input `%0` for the duration of the use `%1`.

A `convert_escape_to_noescape [not_guaranteed] %opd` indicates that the lifetime of its operand was not guaranteed by SILGen and a mandatory pass must be run to ensure the lifetime of `%opd`` for the conversion's uses.

A `convert_escape_to_noescape [escaped]` indicates that the result was passed to a function (materializeForSet) which escapes the closure in a way not expressed by the convert's users. The mandatory pass must ensure the lifetime in a conservative way.

classify_bridge_object

```
sil-instruction ::= 'classify_bridge_object' sil-operand

%1 = classify_bridge_object %0 : $Builtin.BridgeObject
// %1 will be of type (Builtin.Int1, Builtin.Int1)
```

Decodes the bit representation of the specified `Builtin.BridgeObject` value, returning two bits: the first indicates whether the object is an Objective-C object, the second indicates whether it is an Objective-C tagged pointer value.

value_to_bridge_object

```
sil-instruction ::= 'value_to_bridge_object' sil-operand

%1 = value_to_bridge_object %0 : $T
// %1 will be of type Builtin.BridgeObject
```

Sets the BridgeObject to a tagged pointer representation holding its operands by tagging and shifting the operand if needed:

```
value_to_bridge_object %x ===
(x << _swift_abi_ObjCReservedLowBits) | _swift_BridgeObject_TaggedPointerBits
```

`%x` thus must not be using any high bits shifted away or the tag bits post-shift. ARC operations on such tagged values are NOPs.

ref_to_bridge_object

```
sil-instruction ::= 'ref_to_bridge_object' sil-operand, sil-operand

%2 = ref_to_bridge_object %0 : $C, %1 : $Builtin.Word
// %1 must be of reference type $C
// %2 will be of type Builtin.BridgeObject
```

Creates a `Builtin.BridgeObject` that references `%0`, with spare bits in the pointer representation populated by bitwise-OR-ing in the value of `%1`. It is undefined behavior if this bitwise OR operation affects the reference identity of `%0`; in other words, after the following instruction sequence:

```
%b = ref_to_bridge_object %r : $C, %w : $Builtin.Word
%r2 = bridge_object_to_ref %b : $Builtin.BridgeObject to $C
```

`%r` and `%r2` must be equivalent. In particular, it is assumed that retaining or releasing the `BridgeObject` is equivalent to retaining or releasing the original reference, and that the above `ref_to_bridge_object` / `bridge_object_to_ref` round-trip can be folded away to a no-op.

On platforms with ObjC interop, there is additionally a platform-specific bit in the pointer representation of a `BridgeObject` that is reserved to indicate whether the referenced object has native Swift refcounting. It is undefined behavior to set this bit when the first operand references an Objective-C object.

bridge_object_to_ref

```
sil-instruction ::= 'bridge_object_to_ref' sil-operand 'to' sil-type

%1 = bridge_object_to_ref %0 : $Builtin.BridgeObject to $C
// $C must be a reference type
// %1 will be of type $C
```

Extracts the object reference from a `Builtin.BridgeObject`, masking out any spare bits.

bridge_object_to_word

```
sil-instruction ::= 'bridge_object_to_word' sil-operand 'to' sil-type

%1 = bridge_object_to_word %0 : $Builtin.BridgeObject to $Builtin.Word
// %1 will be of type $Builtin.Word
```

Provides the bit pattern of a `Builtin.BridgeObject` as an integer.

thin_to_thick_function

```
sil-instruction ::= 'thin_to_thick_function' sil-operand 'to' sil-type

%1 = thin_to_thick_function %0 : $@convention(thin) T -> U to $T -> U
// %0 must be of a thin function type $@convention(thin) T -> U
// The destination type must be the corresponding thick function type
// %1 will be of type $T -> U
```

Converts a thin function value, that is, a bare function pointer with no context information, into a thick function value with ignored context. Applying the resulting thick function value is equivalent to applying the original thin value. The `thin_to_thick_function` conversion may be eliminated if the context is proven not to be needed.

thick_to_objc_metatype

```
sil-instruction ::= 'thick_to_objc_metatype' sil-operand 'to' sil-type

%1 = thick_to_objc_metatype %0 : $@thick T.Type to $@objc_metatype T.Type
// %0 must be of a thick metatype type $@thick T.Type
// The destination type must be the corresponding Objective-C metatype type
// %1 will be of type $@objc_metatype T.Type
```

Converts a thick metatype to an Objective-C class metatype. `T` must be of class, class protocol, or class protocol composition type.

objc_to_thick_metatype

```
sil-instruction ::= 'objc_to_thick_metatype' sil-operand 'to' sil-type

%1 = objc_to_thick_metatype %0 : $@objc_metatype T.Type to $@thick T.Type
// %0 must be of an Objective-C metatype type $@objc_metatype T.Type
// The destination type must be the corresponding thick metatype type
// %1 will be of type $@thick T.Type
```

Converts an Objective-C class metatype to a thick metatype. `T` must be of class, class protocol, or class protocol composition type.

objc_metatype_to_object

TODO

objc_existential_metatype_to_object

TODO

Checked Conversions

Some user-level cast operations can fail and thus require runtime checking.

The unconditional_checked_cast_addr, unconditional_checked_cast_value and unconditional_checked_cast instructions performs an unconditional checked cast; it is a runtime failure if the cast fails. The checked_cast_addr_br, checked_cast_value_br and checked_cast_br terminator instruction performs a conditional checked cast; it branches to one of two destinations based on whether the cast succeeds or not.

unconditional_checked_cast

```
sil-instruction ::= 'unconditional_checked_cast' sil-operand 'to' sil-type

%1 = unconditional_checked_cast %0 : $A to $B
%1 = unconditional_checked_cast %0 : $*A to $*B
// $A and $B must be both objects or both addresses
// %1 will be of type $B or $*B
```

Performs a checked scalar conversion, causing a runtime failure if the conversion fails. Casts that require changing representation or ownership are unsupported.

unconditional_checked_cast_addr

```
sil-instruction ::= 'unconditional_checked_cast_addr'
                    sil-type 'in' sil-operand 'to'
                    sil-type 'in' sil-operand

unconditional_checked_cast_addr $A in %0 : $*@thick A to $B in %1 : $*@thick B
// $A and $B must be both addresses
// %1 will be of type $*B
// $A is destroyed during the conversion. There is no implicit copy.
```

Performs a checked indirect conversion, causing a runtime failure if the conversion fails.

unconditional_checked_cast_value

```
sil-instruction ::= 'unconditional_checked_cast_value'
                    sil-operand 'to' sil-type

%1 = unconditional_checked_cast_value %0 : $A to $B
// $A must not be an address
// $B must not be an address
// %1 will be of type $B
// $A is destroyed during the conversion. There is no implicit copy.
```

Performs a checked conversion, causing a runtime failure if the conversion fails. Unlike *unconditional_checked_cast*, this destroys its operand and creates a new value. Consequently, this supports bridging objects to values, as well as casting to a different ownership classification such as *$AnyObject* to *$T.Type*.

Runtime Failures

cond_fail

```
sil-instruction ::= 'cond_fail' sil-operand, string-literal

cond_fail %0 : $Builtin.Int1, "failure reason"
// %0 must be of type $Builtin.Int1
```

This instruction produces a runtime failure if the operand is one. Execution proceeds normally if the operand is zero. The second operand is a static failure message, which is displayed by the debugger in case the failure is triggered.

Terminators

These instructions terminate a basic block. Every basic block must end with a terminator. Terminators may only appear as the final instruction of a basic block.

unreachable

```
sil-terminator ::= 'unreachable'

unreachable
```

Indicates that control flow must not reach the end of the current basic block. It is a dataflow error if an unreachable terminator is

reachable from the entry point of a function and is not immediately preceded by an `apply` of a no-return function.

return

```
sil-terminator ::= 'return' sil-operand

return %0 : $T
// $T must be the return type of the current function
```

Exits the current function and returns control to the calling function. If the current function was invoked with an `apply` instruction, the result of that function will be the operand of this `return` instruction. If the current function was invoked with a `try_apply` instruction, control resumes at the normal destination, and the value of the basic block argument will be the operand of this `return` instruction.

If the current function is a `yield_once` coroutine, there must not be a path from the entry block to a `return` which does not pass through a `yield` instruction. This rule does not apply in the `raw` SIL stage.

`return` does not retain or release its operand or any other values.

A function must not contain more than one `return` instruction.

throw

```
sil-terminator ::= 'throw' sil-operand

throw %0 : $T
// $T must be the error result type of the current function
```

Exits the current function and returns control to the calling function. The current function must have an error result, and so the function must have been invoked with a `try_apply` instruction. Control will resume in the error destination of that instruction, and the basic block argument will be the operand of the `throw`.

`throw` does not retain or release its operand or any other values.

A function must not contain more than one `throw` instruction.

yield

```
sil-terminator ::= 'yield' sil-yield-values
                     ',' 'resume' sil-identifier
                     ',' 'unwind' sil-identifier
sil-yield-values ::= sil-operand
sil-yield-values ::= '(' (sil-operand (',' sil-operand)*)? ')'
```

Temporarily suspends the current function and provides the given values to the calling function. The current function must be a coroutine, and the yield values must match the yield types of the coroutine. If the calling function resumes the coroutine normally, control passes to the `resume` destination. If the calling function aborts the coroutine, control passes to the `unwind` destination.

The `resume` and `unwind` destination blocks must be uniquely referenced by the `yield` instruction. This prevents them from becoming critical edges.

In a `yield_once` coroutine, there must not be a control flow path leading from the `resume` edge to another `yield` instruction in this function. This rule does not apply in the `raw` SIL stage.

There must not be a control flow path leading from the `unwind` edge to a `return` instruction, to a `throw` instruction, or to any block reachable from the entry block via a path that does not pass through an `unwind` edge. That is, the blocks reachable from `unwind` edges must jointly form a disjoint subfunction of the coroutine.

unwind

```
sil-terminator ::= 'unwind'
```

Exits the current function and returns control to the calling function, completing an unwind from a `yield`. The current function must be a coroutine.

`unwind` is only permitted in blocks reachable from the `unwind` edges of `yield` instructions.

br

```
sil-terminator ::= 'br' sil-identifier
                     '(' (sil-operand (',' sil-operand)*)? ')'

br label (%0 : $A, %1 : $B, ...)
// `label` must refer to a basic block label within the current function
// %0, %1, etc. must be of the types of `label`'s arguments
```

Unconditionally transfers control from the current basic block to the block labeled `label`, binding the given values to the arguments of the destination basic block.

cond_br

```
sil-terminator ::= 'cond_br' sil-operand ','
                     sil-identifier '(' (sil-operand (',' sil-operand)*)? ')' ','
                     sil-identifier '(' (sil-operand (',' sil-operand)*)? ')'

cond_br %0 : $Builtin.Int1, true_label (%a : $A, %b : $B, ...), \
                            false_label (%x : $X, %y : $Y, ...)
// %0 must be of $Builtin.Int1 type
// `true_label` and `false_label` must refer to block labels within the
//   current function and must not be identical
// %a, %b, etc. must be of the types of `true_label`'s arguments
// %x, %y, etc. must be of the types of `false_label`'s arguments
```

Conditionally branches to `true_label` if `%0` is equal to `1` or to `false_label` if `%0` is equal to `0`, binding the corresponding set of values to the arguments of the chosen destination block.

switch_value

```
sil-terminator ::= 'switch_value' sil-operand
                     (',' sil-switch-value-case)*
                     (',' sil-switch-default)?
sil-switch-value-case ::= 'case' sil-value ':' sil-identifier
sil-switch-default ::= 'default' sil-identifier

switch_value %0 : $Builtin.Int<n>, case %1: label1, \
                                   case %2: label2, \
                                   ...,             \
                                   default labelN

// %0 must be a value of builtin integer type $Builtin.Int<n>
// `label1` through `labelN` must refer to block labels within the current
//   function
// FIXME: All destination labels currently must take no arguments
```

Conditionally branches to one of several destination basic blocks based on a value of builtin integer or function type. If the operand value matches one of the `case` values of the instruction, control is transferred to the corresponding basic block. If there is a `default` basic block, control is transferred to it if the value does not match any of the `case` values. It is undefined behavior if the value does not match any cases and no `default` branch is provided.

select_value

```
sil-instruction ::= 'select_value' sil-operand sil-select-value-case*
                    (',' 'default' sil-value)?
                    ':' sil-type
```

```
sil-select-value-case ::= 'case' sil-value ':' sil-value


%n = select_value %0 : $U, \
  case %c1: %r1,           \
  case %c2: %r2, /* ... */ \
  default %r3 : $T

// $U must be a builtin type. Only integers types are supported currently.
// c1, c2, etc must be of type $U
// %r1, %r2, %r3, etc. must have type $T
// %n has type $T
```

Selects one of the "case" or "default" operands based on the case of a value. This is equivalent to a trivial switch_value branch sequence:

```
entry:
  switch_value %0 : $U,            \
    case %c1: bb1,           \
    case %c2: bb2, /* ... */ \
    default bb_default
bb1:
  br cont(%r1 : $T) // value for %c1
bb2:
  br cont(%r2 : $T) // value for %c2
bb_default:
  br cont(%r3 : $T) // value for default
cont(%n : $T):
  // use argument %n
```

but turns the control flow dependency into a data flow dependency.

switch_enum

```
sil-terminator ::= 'switch_enum' sil-operand
                     (',' sil-switch-enum-case)*
                     (',' sil-switch-default)?
sil-switch-enum-case ::= 'case' sil-decl-ref ':' sil-identifier

switch_enum %0 : $U, case #U.Foo!enumelt: label1, \
                      case #U.Bar!enumelt: label2, \
                      ...,                 \
                      default labelN

// %0 must be a value of enum type $U
// #U.Foo, #U.Bar, etc. must be 'case' declarations inside $U
// `label1` through `labelN` must refer to block labels within the current
//   function
// label1 must take either no basic block arguments, or a single argument
//   of the type of #U.Foo's data
// label2 must take either no basic block arguments, or a single argument
//   of the type of #U.Bar's data, etc.
// labelN must take no basic block arguments
```

Conditionally branches to one of several destination basic blocks based on the discriminator in a loadable `enum` value. Unlike `switch_int`, `switch_enum` requires coverage of the operand type: If the `enum` type cannot be switched exhaustively in the current function, the `default` branch is required; otherwise, the `default` branch is required unless a destination is assigned to every `case` of the `enum`. The destination basic block for a `case` may take an argument of the corresponding `enum` `case`'s data type (or of the address type, if the operand is an address). If the branch is taken, the destination's argument will be bound to the associated data inside the original enum value. For example:

```
enum Foo {
  case Nothing
  case OneInt(Int)
  case TwoInts(Int, Int)
}

sil @sum_of_foo : $Foo -> Int {
entry(%x : $Foo):
  switch_enum %x : $Foo,       \
    case #Foo.Nothing!enumelt: nothing, \
    case #Foo.OneInt!enumelt:  one_int, \
    case #Foo.TwoInts!enumelt: two_ints

nothing:
  %zero = integer_literal $Int, 0
  return %zero : $Int

one_int(%y : $Int):
  return %y : $Int

two_ints(%ab : $(Int, Int)):
  %a = tuple_extract %ab : $(Int, Int), 0
  %b = tuple_extract %ab : $(Int, Int), 1
  %add = function_ref @add : $(Int, Int) -> Int
  %result = apply %add(%a, %b) : $(Int, Int) -> Int
  return %result : $Int
}
```

On a path dominated by a destination block of `switch_enum`, copying or destroying the basic block argument has equivalent reference counting semantics to copying or destroying the `switch_enum` operand:

```
  // This retain_value...
  retain_value %e1 : $Enum
  switch_enum %e1, case #Enum.A: a, case #Enum.B: b
a(%a : $A):
  // ...is balanced by this release_value
  release_value %a
b(%b : $B):
  // ...and this one
  release_value %b
```

switch_enum_addr

```
sil-terminator ::= 'switch_enum_addr' sil-operand
                     (',' sil-switch-enum-case)*
                     (',' sil-switch-default)?

switch_enum_addr %0 : $*U, case #U.Foo!enumelt: label1, \
                                      case #U.Bar!enumelt: label2, \
                                      ...,                 \
                                      default labelN

// %0 must be the address of an enum type $*U
// #U.Foo, #U.Bar, etc. must be cases of $U
// `label1` through `labelN` must refer to block labels within the current
//   function
// The destinations must take no basic block arguments
```

Conditionally branches to one of several destination basic blocks based on the discriminator in the enum value referenced by the address operand.

Unlike `switch_int`, `switch_enum` requires coverage of the operand type: If the `enum` type cannot be switched exhaustively in the current function, the `default` branch is required; otherwise, the `default` branch is required unless a destination is assigned to every `case` of the `enum`. Unlike `switch_enum`, the payload value is not passed to the destination basic blocks; it must be projected out separately with unchecked_take_enum_data_addr.

dynamic_method_br

```
sil-terminator ::= 'dynamic_method_br' sil-operand ',' sil-decl-ref
                     ',' sil-identifier ',' sil-identifier

dynamic_method_br %0 : $P, #X.method, bb1, bb2
// %0 must be of protocol type
// #X.method must be a reference to an @objc method of any class
// or protocol type
```

Looks up the implementation of an Objective-C method with the same selector as the named method for the dynamic type of the value inside an existential container. The "self" operand of the result function value is represented using an opaque type, the value for which must be projected out as a value of type `Builtin.ObjCPointer`.

If the operand is determined to have the named method, this instruction branches to `bb1`, passing it the uncurried function corresponding to the method found. If the operand does not have the named method, this instruction branches to `bb2`.

checked_cast_br

```
sil-terminator ::= 'checked_cast_br' sil-checked-cast-exact?
                   sil-operand 'to' sil-type ','
                   sil-identifier ',' sil-identifier
sil-checked-cast-exact ::= '[' 'exact' ']'

checked_cast_br %0 : $A to $B, bb1, bb2
checked_cast_br %0 : $*A to $*B, bb1, bb2
checked_cast_br [exact] %0 : $A to $A, bb1, bb2
// $A and $B must be both object types or both address types
// bb1 must take a single argument of type $B or $*B
// bb2 must take no arguments
```

Performs a checked scalar conversion from `$A` to `$B`. If the conversion succeeds, control is transferred to `bb1`, and the result of the cast is passed into `bb1` as an argument. If the conversion fails, control is transferred to `bb2`.

An exact cast checks whether the dynamic type is exactly the target type, not any possible subtype of it. The source and target types must be class types.

checked_cast_value_br

```
sil-terminator ::= 'checked_cast_value_br'
                   sil-operand 'to' sil-type ','
                   sil-identifier ',' sil-identifier
sil-checked-cast-exact ::= '[' 'exact' ']'

checked_cast_value_br %0 : $A to $B, bb1, bb2
// $A must be not be an address
// $B must be an opaque value
// bb1 must take a single argument of type $B
// bb2 must take no arguments
```

Performs a checked opaque conversion from `$A` to `$B`. If the conversion succeeds, control is transferred to `bb1`, and the result of the cast is passed into `bb1` as an argument. If the conversion fails, control is transferred to `bb2`.

checked_cast_addr_br

```
sil-terminator ::= 'checked_cast_addr_br'
                   sil-cast-consumption-kind
                   sil-type 'in' sil-operand 'to'
                   sil-stype 'in' sil-operand ','
                   sil-identifier ',' sil-identifier
sil-cast-consumption-kind ::= 'take_always'
sil-cast-consumption-kind ::= 'take_on_success'
sil-cast-consumption-kind ::= 'copy_on_success'

checked_cast_addr_br take_always $A in %0 : $*@thick A to $B in %2 : $*@thick B, bb1, bb2
// $A and $B must be both address types
// bb1 must take a single argument of type $*B
// bb2 must take no arguments
```

Performs a checked indirect conversion from `$A` to `$B`. If the conversion succeeds, control is transferred to `bb1`, and the result of the cast is left in the destination. If the conversion fails, control is transferred to `bb2`.

try_apply

```
sil-terminator ::= 'try_apply' sil-value
                     sil-apply-substitution-list?
                     '(' (sil-value (',' sil-value)*)? ')'
                     ':' sil-type
  'normal' sil-identifier, 'error' sil-identifier

try_apply %0(%1, %2, ...) : $(A, B, ...) -> (R, @error E),
  normal bb1, error bb2
bb1(%3 : R):
bb2(%4 : E):

// Note that the type of the callee '%0' is specified *after* the arguments
// %0 must be of a concrete function type $(A, B, ...) -> (R, @error E)
// %1, %2, etc. must be of the argument types $A, $B, etc.
```

Transfers control to the function specified by `%0`, passing it the given arguments. When `%0` returns, control resumes in either the normal destination (if it returns with `return`) or the error destination (if it returns with `throw`).

`%0` must have a function type with an error result.

The rules on generic substitutions are identical to those of `apply`.

await_async_continuation

```
sil-terminator ::= 'await_async_continuation' sil-value
                     ',' 'resume' sil-identifier
                     (',' 'error' sil-identifier)?

await_async_continuation %0 : $UnsafeContinuation<T>, resume bb1
await_async_continuation %0 : $UnsafeThrowingContinuation<T>, resume bb1, error bb2

bb1(%1 : @owned $T):
bb2(%2 : @owned $Error):
```

Suspends execution of an `@async` function until the continuation is resumed. The continuation must be the result of a `get_async_continuation` or `get_async_continuation_addr` instruction within the same function; see the documentation for `get_async_continuation` for discussion of further constraints on the IR between `get_async_continuation[_addr]` and `await_async_continuation`. This terminator can only appear inside an `@async` function. The instruction must always have a `resume` successor, but must have an `error` successor if and only if the operand is an `UnsafeThrowingContinuation<T>`.

If the operand is the result of a `get_async_continuation` instruction, then the `resume` successor block must take an argument whose type is the maximally-abstracted lowered type of `T`, matching the type argument of the `Unsafe[Throwing]Continuation<T>` operand. The value of the `resume` argument is owned by the current function. If the operand is the result of a `get_async_continuation_addr` instruction, then the `resume` successor block must *not* take an argument; the resume value will be written to the memory referenced by the operand to the `get_async_continuation_addr` instruction, after which point the value in that memory becomes owned by the current function. With either variant, if the `await_async_continuation` instruction has an `error` successor block, the `error` block must take a single `Error` argument, and that argument is owned by the enclosing function. The memory referenced by a `get_async_continuation_addr` instruction remains uninitialized when `await_async_continuation` resumes on the `error` successor.

It is possible for a continuation to be resumed before `await_async_continuation`. In this case, the resume operation returns

immediately to its caller. When the `await_async_continuation` instruction later executes, it then immediately transfers control to its `resume` or `error` successor block, using the resume or error value that the continuation was already resumed with.

Differentiable Programming

differentiable_function

```
sil-instruction ::= 'differentiable_function'
                    sil-differentiable-function-parameter-indices
                    sil-value ':' sil-type
                    sil-differentiable-function-derivative-functions-clause?

sil-differentiable-function-parameter-indices ::=
    '[' 'parameters' [0-9]+ (' ' [0-9]+)* ']'
sil-differentiable-derivative-functions-clause ::=
    'with_derivative'
    '{' sil-value ':' sil-type ',' sil-value ':' sil-type '}'

differentiable_function [parameters 0] %0 : $(T) -> T \
  with_derivative {%1 : $(T) -> (T, (T) -> T), %2 : $(T) -> (T, (T) -> T)}
```

Creates a `@differentiable` function from an original function operand and derivative function operands (optional). There are two derivative function kinds: a Jacobian-vector products (JVP) function and a vector-Jacobian products (VJP) function.

`[parameters ...]` specifies parameter indices that the original function is differentiable with respect to.

The `with_derivative` clause specifies the derivative function operands associated with the original function.

The differentiation transformation canonicalizes all *differentiable_function* instructions, generating derivative functions if necessary to fill in derivative function operands.

In raw SIL, the `with_derivative` clause is optional. In canonical SIL, the `with_derivative` clause is mandatory.

linear_function

```
sil-instruction ::= 'linear_function'
                    sil-linear-function-parameter-indices
                    sil-value ':' sil-type
                    sil-linear-function-transpose-function-clause?

sil-linear-function-parameter-indices ::=
    '[' 'parameters' [0-9]+ (' ' [0-9]+)* ']'
sil-linear-transpose-function-clause ::=
    with_transpose sil-value ':' sil-type

linear_function [parameters 0] %0 : $(T) -> T with_transpose %1 : $(T) -> T
```

Bundles a function with its transpose function into a `@differentiable(_linear)` function.

`[parameters ...]` specifies parameter indices that the original function is linear with respect to.

A `with_transpose` clause specifies the transpose function associated with the original function. When a `with_transpose` clause is not specified, the mandatory differentiation transform will add a `with_transpose` clause to the instruction.

In raw SIL, the `with_transpose` clause is optional. In canonical SIL, the `with_transpose` clause is mandatory.

differentiable_function_extract

```
sil-instruction ::= 'differentiable_function_extract'
                    '[' sil-differentiable-function-extractee ']'
                    sil-value ':' sil-type
                    ('as' sil-type)?

sil-differentiable-function-extractee ::= 'original' | 'jvp' | 'vjp'

differentiable_function_extract [original] %0 : $@differentiable (T) -> T
differentiable_function_extract [jvp] %0 : $@differentiable (T) -> T
differentiable_function_extract [vjp] %0 : $@differentiable (T) -> T
differentiable_function_extract [jvp] %0 : $@differentiable (T) -> T \
  as $(@in_constant T) -> (T, (T.TangentVector) -> T.TangentVector)
```

Extracts the original function or a derivative function from the given `@differentiable` function. The extractee is one of the following: `[original]`, `[jvp]`, or `[vjp]`.

In lowered SIL, an explicit extractee type may be provided. This is currently used by the LoadableByAddress transformation, which rewrites function types.

linear_function_extract

```
sil-instruction ::= 'linear_function_extract'
                    '[' sil-linear-function-extractee ']'
                    sil-value ':' sil-type

sil-linear-function-extractee ::= 'original' | 'transpose'

linear_function_extract [original] %0 : $@differentiable(_linear) (T) -> T
linear_function_extract [transpose] %0 : $@differentiable(_linear) (T) -> T
```

Extracts the original function or a transpose function from the given `@differentiable(_linear)` function. The extractee is one of the following: `[original]` or `[transpose]`.

differentiability_witness_function

```
sil-instruction ::=
    'differentiability_witness_function'
    '[' sil-differentiability-witness-function-kind ']'
    '[' differentiability-kind ']'
    '[' 'parameters' sil-differentiability-witness-function-index-list ']'
    '[' 'results' sil-differentiability-witness-function-index-list ']'
    generic-parameter-clause?
    sil-function-name ':' sil-type

sil-differentiability-witness-function-kind ::= 'jvp' | 'vjp' | 'transpose'
sil-differentiability-witness-function-index-list ::= [0-9]+ (' ' [0-9]+)*

differentiability_witness_function [vjp] [reverse] [parameters 0] [results 0] \
  <T where T: Differentiable> @foo : $(T) -> T
```

Looks up a differentiability witness function (JVP, VJP, or transpose) for a referenced function via SIL differentiability witnesses.

The differentiability witness function kind identifies the witness function to look up: `[jvp]`, `[vjp]`, or `[transpose]`.

The remaining components identify the SIL differentiability witness:

- Original function name.
- Differentiability kind.
- Parameter indices.
- Result indices.
- Witness generic parameter clause (optional). When parsing SIL, the parsed witness generic parameter clause is combined with the original function's generic signature to form the full witness generic signature.

Assertion configuration

To be able to support disabling assertions at compile time there is a builtin `assertion_configuration` function. A call to this function can be replaced at compile time by a constant or can stay opaque.

All calls to the `assert_configuration` function are replaced by the constant propagation pass to the appropriate constant

depending on compile time settings. Subsequent passes remove dependent unwanted control flow. Using this mechanism we support conditionally enabling/disabling of code in SIL libraries depending on the assertion configuration selected when the library is linked into user code.

There are three assertion configurations: Debug (0), Release (1) and DisableReplacement (-1).

The optimization flag or a special assert configuration flag determines the value. Depending on the configuration value assertions in the standard library will be executed or not.

The standard library uses this builtin to define an assert that can be disabled at compile time.

```
func assert(...) {
  if (Int32(Builtin.assert_configuration()) == 0) {
    _fatal_error_message(message, ...)
  }
}
```

The `assert_configuration` function application is serialized when we build the standard library (we recognize the `-parse-stdlib` option and don't do the constant replacement but leave the function application to be serialized to sil).

The compiler flag that influences the value of the `assert_configuration` function application is the optimization flag: at `-Onone` the application will be replaced by `Debug` at higher optimization levels the instruction will be replaced by `Release`. Optionally, the value to use for replacement can be specified with the `-assert-config` flag which overwrites the value selected by the optimization flag (possible values are `Debug`, `Release`, `DisableReplacement`).

If the call to the `assert_configuration` function stays opaque until IRGen, IRGen will replace the application by the constant representing Debug mode (0). This happens we can build the standard library .dylib. The generate sil will retain the function call but the generated .dylib will contain code with assertions enabled.