# SIL Function Conventions

Throughout the compiler, integer indices are used to identify argument positions in several different contexts:

- A `SILFunctionType` has a tuple of parameters.
- The SIL function definition has a list of `SILFunctionArgument`. This is the callee-side argument list. It includes indirect results.
- `apply`, `try_apply`, and `begin_apply` have "applied arguments": the subset of instruction operands representing the callee's `SILFunctionArgument` list.
- `partial_apply` has "applied arguments": the subset of instruction operands representing closure captures. Closure captures in turn map to a subset of the callee's `SILFunctionArgument` list.
- In the above three contexts, `SILFunctionArgument`, `apply`, and `partial_apply`, the argument indices also depend on the SIL stage: Canonical vs. Lowered.

Consider the example:

```
func example<T>(i: Int, t: T) -> (Int, T) {
  let foo = { return ($0, t) }
  return foo(i)
}
```

The closure `foo` has the indices as numbered below in each context, ignoring the calling convention and boxing/unboxing of captures for brevity:

The closure's `SILFunctionType` has two direct formal parameters at indices ( `#0`, `#1` ) and one direct formal result of tuple type:

```
SILFunctionType(foo): (#0: Int, #1: T) -> @out (Int, T)
```

Canonical SIL with opaque values matches `SILFunctionType`. The definition of `foo` has two direct `SILFunctionArgument`s at ( `#0`, `#1` ):

```
SILFunctionArguments: (#0: Int, #1: T) -> (Int, T)
```

The Lowered SIL for `foo`s definition has an indirect "result argument" at index #0. The function parameter indices are now ( `#1`, `#2` ):

```
SILFunctionArguments: (#0: *T, #1: Int, #2: T) -> Int
```

Creation of the closure has one applied argument at index `#0`. Note that the first applied argument is actually the second operand (the first is the callee), and in lowered SIL, it is actually the third SILFunctionArgument (after the indirect result and first parameter):

```
%closure = partial_apply @foo(#0: t)
```

Application of the closure with opaque values has one applied argument:

```
%resultTuple = apply %closure(#0: i)
```

Lowered application of the closure has two applied arguments:

```
%directResult = apply %closure(#0: %indirectResult: *T, #1: i)
```

The mapping between `SILFunctionType` and `SILFunctionArgument`, which depends on the SIL stage, is managed by the SILFunctionConventions abstraction. This API follows naming conventions to communicate the meaning of the integer indices:

- "Parameters" refer to the function signature's tuple of arguments.
- "SILArguments" refer to the set of `SILFunctionArgument` in the callee's entry block, including any indirect results required by the current SIL stage.

These argument indices and their relative offsets should never be hardcoded. Although this is common practice in LLVM, it should be avoided in SIL: (a) the structure of SIL instructions, particularly applies, is much more nuanced than LLVM IR, (b) assumptions that may be valid at the initial point of use are often copied into parts of the code where they are no longer valid; and (c) unlike LLVM IR, SIL is not stable and will continue evolving.

Translation between SILArgument and parameter indices should use: `SILFunctionConventions::getSILArgIndexOfFirstParam()`.

Translation between SILArgument and result indices should use: `SILFunctionConventions::getSILArgIndexOfFirstIndirectResult()`.

Convenience methods exist for the most common uses, so there is typically no need to use the above "IndexOfFirst" methods to translate one integer index into another. The naming convention of the convenience method should clearly indicate which form of index it expects. For example, information about a parameter's type can be retrieved directly from a SILArgument index: `getParamInfoForSILArg(index)`, `getSILArgumentConvention(index)`, and `getSILArgumentType(index)`.

Another abstraction, `ApplySite`, abstracts over the various kinds of `apply` instructions, including `try_apply`, `begin_apply`, and `partial_apply`.

`ApplySite::getSubstCalleeConv()` is commonly used to query the callee's `SILFunctionConventions` which provides information about the function's type and its definition as explained above. Information about the applied arguments can be queried directly from the `ApplySite` API.

For example, `ApplySite::getAppliedArgumentConvention(index)` takes an applied argument index, while `SILFunctionArguments::getSILArgumentConvention(index`) takes a `SILFunctionArgument` index. They both return the same information, but from a different viewpoint.

A common mistake is to directly map the ApplySite's caller-side arguments onto callee-side SILFunctionArguments. This happens to work until the same code is exposed to a `partial_apply`. Instead, use the `ApplySite` API for applied argument indices, or use `ApplySite::getCalleeArgIndexOfFirstAppliedArg()` to translate the apply's arguments into function convention arguments.

Consistent use of common idioms for accessing arguments should be adopted throughout the compiler. Plenty of bugs have resulted from assumptions about the form of SIL made in one area of the compiler that have been copied into other parts of the compiler. For example, knowing that a block of code is guarded by a dynamic condition that rules out PartialApplies is no excuse to conflate applied arguments with function arguments. Also, without consistent use of common idioms, it becomes overly burdensome to evolve these APIs over time.