目錄

Introduction

0x0 摘要

SIL是Swift的中间语言，提供了和Swift编程语言实现相关的高层的语义信息，其形式遵循了SSA（Static-Single-Assignemt）标准。它支持以下的用例：

- 提供高层的guaranteed optimization，提供可预期的运行时优化基线，并在诊断分析中提供可预期的行为。
- 提供诊断分析中的数据流分析通道处理器（Passes），以确保代码符合Swift的强制规范。包括：变量的初始化检查，构造器缺失检查，代码可及性检查，switch分支的条件覆盖分析等。
- 提供高层的编译优化通道，包括：ARC优化，动态方法去虚拟化，闭包内联化，使用堆内存提升为使用栈内存，使用栈内存提升为使用SSA寄存器，聚合内存标量替换（scalar replacement of aggreates，把聚合的大块内存申请分割为多个小块内存申请），泛型函数实例化等。
- 为Swift库提供一种稳定的分发格式，引用端可以使用内联、泛型等特性进行编译优化。

相比LLVM的中间语言，通常SIL是“体系结构无关”的，即它不关心具体输出的目标代码运行在什么架构平台上，但它也可以和LLVM一样做到“target-specific”。

了解更多关于开发SIL和SIL通道处理器的信息，参见《SILProgrammersManual》

GOSSARY

guaranteed optimization | | 一种基于证据的可被检验的编译器优化技术 pass ｜ 通道处理器 ｜ 可以理解为单向的通道处理器 scalar replacement of aggreates｜ 聚合内存标量替换 ｜一种优化，把聚合的大块内存申请分割为多个小块内存申请 target-specific ｜ 指定目标产物｜ 指定输出的产物为某平台或体系结构所支持的格式

0x1 SIL和Swift编译器

在高层设计中，swift编译器遵循严格的管道设计架构：

- 源码输入 *Parser* 模块进行解析处理，输出构造好的AST抽象语法树
- AST输入 *Sema* 模块进行语义分析和类型检查，进一步标注更丰富的类型信息，输出AST‘
- AST’输入 *SILGen* 模块生成 *raw SIL*
- *raw SIL* 会输入到一系列 *可检验的优化处理器(Guaranteed Optimization Passes)* 和 *诊断处理器*中进行优化，输出可能的警告和错误，以及 *cannonical SIL* 。 注意，以上管道处理器是编译过程的必选项，必然运行
- (可选) *cannonical SIL* 输入到通用管道处理器中进行最终的性能优化。这个过程是可选的，可以开关或者指定特定的优化等级
- 优化后的*cannonical SIL* 输入 *IRGen* 模块，降级转化为LLVM的IR语言
- (可选) 降级后LLVM IR输入到的LLVM后端编译器进行优化处理，并最终生成二进制目标文件。

与SIL处理相关的特定阶段如下：

SILGen

SILGen输入已确认类型的AST',通过遍历AST'，生成 *raw SIL* 。 *raw SIL* 具有以下特点：

- 变量的load和store操作使用可变内存位置，没有使用严格的SSA形式。这有点像最早Clang等前端编译器输出的alloca-heavy(大量使用alloc)的LLVM IR。 另一方面，Swift把变量包裹在一个 "boxes"的容器里，对这个通用结构进行引用计数管理，同时也作为闭包截获外部变量的容器。
- 数据流的一些必要检查不在这个阶段执行。包括声明时赋值（definitive assignment），函数返回检查，switch条件覆盖率（TBD）等。
- 透明函数(transparent function,强制函数内联优化的输出)优化不在这个阶段

Guaranteed Optimization and Diagnostic Passes

在SILGen生成 *raw sil* ，输入到必选的优化通道处理器中，随后输入诊断通道处理器中。如果源码不变，我们不希望因为编译器的升级或改动导致在该阶段输出的错误、警告等诊断信息跟着发生变化，因此这要求这些通道处理器都设计得足够简单，并且可以预测输出结果。

- **强制内联化(Mandatory inlining)** 也称为透明函数
- **内存提升（Memory Promotion）** 是指把内存使用的方式优化提升为更高效的方式，该处理有两个阶段：第一阶段是把 `alloc_box` 的堆内存申请指令提升为 `alloc_stack` 栈内存； 第二阶段是把地址非暴露方式的 `alloc_stack` 栈内存方式提升为使用 SSA寄存器的方式。

- **常量折叠（Constant Folding）** 把常量或表达式不断地简化； **Constant propagation 常量传递** 把常数的值替换到表达式中，从而使得代码可以进一步优化。
- **返回分析（Return analysis）** 确认代码的每一段逻辑分支都能符合函数原型的定义。对于有返回类型的函数，确保每个分支的执行末端都会有正确类型的返回；对于不返回的函数，则确保每个分支都不会返回；否则，报错。
- **临界拆分（Critical edge splitting）** 是指把多个BB(BasicBlock)拆开，主要针对的是“非cond_branch”的terminator的BB

如果以上的诊断都通过了，那么会输出Canonical SIL代码。

TODO：

- 泛型特化
- 基本ARC优化，指为了达到可接受的性能而进行的最基本的，而且是强制的ARC优化，即使在-Onone模式也会执行

General Optimization Passes

SIL可以提供更多和语言特性相关的类型信息，从而可以在高层级(high-level)进行优化。而LLVM IR不适合做这类优化，因为它是通用的，和高级语言特性无关的。这也是SIL存在的意义。

- **泛型特化（Generic Specialization）**：分析泛型函数的特化调用，并生成新的特化版本的函数。然后，把泛型的所有特化调用的地方重写成对特化函数的直接调用。
- **Witness 和 VTable 反虚拟化技术**通过查找相关class的vtable或者特定类型的vitness table，将一个特定类型的方法调用替换为查表得到的函数。
- **性能优化内联**：为了提升执行效率，强制把可内联化的内联化。
- **引用计数优化**
- **内存提升/内存优化**
- **High-level domain specific optimizations** 配合标准库进行的高级优化，比如对Swift容器类型（Array 、String等）进行高层级的优化，具体见
  `HighLevelSILOptimizations <HighLevelSILOptimizations.rst>` _

```
FYI:
Constant folding
Constant propagation
ref: https://zh.wikipedia.org/wiki/%E5%B8%B8%E6%95%B8%E6%8A%98%E7%96%8A


BasicBlock
由顺序执行的指令组成，是串型且不分叉的执行过程，是SIL程序的执行片段和组织单位。
可以理解为拆分开的子程序


terminator
是SIL一类特殊的指令，是每一个BB的最后一条指令，表明了这段BB该如何结束。
cond_branch是具体的一个terminator指令，指条件分支结束指令，会接受一个参数，根据参数决定控制
```

0x2 Syntax 句法

- SIL的句法是Swift句法的扩展，因为SIL依赖于Swift的类型系统以及声明(Declarations)。.sil文件就是Swift源码再加上扩展的SIL定义。
- 在句法解析过程中，仅解析Swift源码的各种声明；Swift的func函数体（body）部分（除了函数内声明的内部函数）以及顶层的代码会被SIL的句法解析忽略。
- .sil文件要求必须显式地引入外部依赖，包括标准库。

```
F.Y.I.
在Swift AST类型系统中，顶层的代码如果没有main函数，则会自动生成TopLevelCodeDecl的节点，并
```

以下是一个 .sil 文件的示例：

```
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 @_Tsoi1pfTSdSd_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 Stages

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

// 同一SIL文件只描述一种Stage
sil-stage ::= 'raw'
sil-stage ::= 'canonical'
```

对于不同的编译阶段，对应了不同的SIL的具体形式，或者说变体的版本。 主要有两个阶段和形式，一个SIL文件的两个阶段互斥：

- **Raw SIL** 这种形式是由SILGen生成，但还未通过guaranteed optimizations和diagnostic passes处理之前的SIL代码。 这时候的Raw SIL可能还未完全构造成SSA图的形式，其数据流可能存在errors。一些指令肯跟还不是canonical的形式，例如，non-address-only类型的assign和destroy_addr指令等。
- **Canonical SIL** 是经过guaranteed optimizations和diagnostic passes处理之后的代码，此时数据流中的errors已经消除，特定的指令也处理成了Canonical的简化形式。基于Canonical SIL 可以进行性能优化并生成最终的目标文件。另外，其也可以作为组件模块分发的一种格式。

SIL Types

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

SIL 类型以$符号开头，SIL的类型系统和Swift的是高度相关的，所以$后面的类型名称大体上是来自于Swift的类型语法。

Type Lowering

类型降级是指把FormalType降级为SILTypes。SILTypes也被称为Lowered Types。FormalType是服务于Swift AST中的类型系统。它的设计意图是抽象和统一句法的表示，比如对所有权转移规范和传参直接性等问题提供一套表示的方案。 另一方面，SIL的设计是为了更好地描述实现细节，从而这也要求SIL有一套不同的类型系统。

需要注意的是，有些情况下的声明是不需要做类型降级的。比如场景：

- 通常属于thin类型的
- 可能有非Swift调用约定的
- 可能在接口中使用bridged类型的
- 可能Ownership转换行为与Swift默认规范不同的

Abstraction Difference

泛型函数具有未绑定类型(uncontrained type)时，必须间接地使用它们。 例如，为未绑定类型开辟足够的内存空间，并传递该内存的地址作为指针使用。

考虑如下的泛型函数：

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

函数参数generator返回一个类型为T的结果，这个结果因为类型是未知的，所以必须将该结果写入一个地址，并通过一个隐式的参数进行传递。 想要处理任意类型的值，其实也并没有其他特别好的替代方案：

- 我们不希望对于每个类型T，都生成一份generateArray的具体实现 // FYI： 类似静态多态
- 我们不想给语言里的每个类型一个通用的表示
- 我们不希望因为T的不同，而导致我们必须动态地构造不同的generator调用

但是，我们也不希望泛型系统的存在给导致非范型代码变得低效。 举个例子，对于函数类型 （）->Int 我们希望能直接返回类型Int；另外，（）->Int是泛型表示 ()->T 的一个合法的类型替换，generateArray < Int >的调用者应该可以直接传入任意的（）->Int类型的参数作为generator。

因此，formaltype在泛型上下文中的表示和在类型替换中的表示会存在差异。我们称之为 *抽象差异（Abstraction Difference）* 。

SIL类型系统的设计希望抽象差异的具体表现能通过SIL类型表达出来。对于嵌套的场景，能用合理抽象的值来表示每一层类型替换。

为达到这个目标，一个泛型实体（generic entity）的formal type做降级的时候，总是用一个抽象模式（abstraction pattern），这个模式由未进行类型替换（unsubstituted）的formal types构成。 例如：

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

例子中，intGen.fn的替换类型用formal type表示就是 () -> Int, 如果把它进行类型降级，则可以得到 @callee_owned () -> Int。 我们可以看到这个函数会直接返回一个Int类型的返回值。

但如果用未进行类型替换的泛型抽象表示我们会得到 () -> T, 对它进行降级，得到 @callee_owned () -> @out Int。 我们可以看到，函数返回类型变成了@out Int，不是直接返回Int了。

在类型降级中使用非约束类型来构成抽象模式，这个模式看起来就像是一个共享的结构，只需要用不同的类型变量填入到相应的可填充类型（materializable type）中。

比如, 如果g的类型是Generator<(Int, Int) -> Float>，那么g.fn在类型在降级的时候，先用(Int, Int) -> Float 来表示T，使用抽象模式是() -> T。 这和使用抽象模式 U -> V是一样。 最终降级的SIL类型表示为：

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

再举个例子，变量h的类型是Generator<(Int, inout Int) -> Float>。其中，类型(Int, inout Int)和 inout Int都不是可填充（materializable）类型，因此他们都不是类型替换的可能的结果。最后h.fn的降级类型如下：

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

通过多层的替换和展开，就达成了用一套SIL Types来表示具有抽象差异（Abstraction Difference）的泛型表示，并进行类型替换和类型降级了。（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负责解析和生成抽象模式（Abstraction Pattern）。

目前，只有function和tuple类型存在抽象差异的问题。

```
FYI
Int 为trivial/loadable类别，可直接通过寄存器传递
而泛型T由于类型未知，是address-only的类别，只能通过指针传递
以上差异导致了调用方式和返回的方式不同

该问题通过reabstraction stub解决，见文章XXX
```

Legal SIL Types

一个合法的SIL的值：

- $T, 表示一个对象类型，其中T是一个合法的loadable type
- $*T, 表示一个地址类型，其中T是一个合法的SIL类型（loadable 或者address-only类型）

一个合法的SIL类型T，包括：

- function，要求满足下列约束
- metatype，用来描述类型的表示（representation）
- tuple，要求其元素合法
- Optional< U >, 要求U为合法的SIL类型
- @box类型，其包裹了一个合法的SIL类型
- 除以上所列，合法的Swift类型（非左值类型）

需要注意的是，在别的递归位置的类型在类型语法上还是formal types。 例如，一个metatype的实例类型或者一个范型的类型参数，它们仍然是formal types，而不是降级的SILTypes.

Address Types

$*T就是$T的指针。可以是一个数据结构的内部指针。可以读写loadable types类型的地址。

$T的T不能是Address-Only类型。Address-Only类型的地址只能通过相关的指令进行操作，比如 copy_addr,destroy_addr，也可以作为函数的参数，但是不能作为AddressType的值。

Address类型不支持引用计数，不像Class的值可以release和retain。

Address类型不是语言的一等(first-class)成员。递归的写法在类型表达式中是非法的，例如$**T是非法的。

地址的地址不能直接地获取。$**T不是一个合法的类型表示。Address类型的值因此不能被allocated，loaded，或者stored（虽然地址是可以的）。

Addresse 类型可以作为函数的indirect入参，但不可作为返回值类型。

Box Types

被捕获的本地变量，以及存储在堆内存上的间接访问值类型(indirect value types)的payloads。

@box T 是一个容器，包裹了一个T类型的可变值。该容器支持引用计数，使用swift-native的引用计数（区别于oc运行时）。因此box类型也可以转成Builtin.NativeObject类型进行访问。

Metatype Types

一个具体的SIL类型，必须说明它的Metatype属性：

- @thin, 该类型不需要额外的外部存储，就可以必要地表达一个具体类型（仅可用于具体的metatypes）
- @thick, 该类型存储个引用，该引用指向一个类型或者该类型的子类（如果是一个具体的class类型的话）
- @objc, 该类型存储个引用，该引用指向一个OC的对象类型（或者它的子类），而不是Swift对象的表示。

Function Types

SIL的function类型和Swift的function类型有多处不同：

- SIL function类型可能是泛型函数。例如，使用function_ref指令访问泛型函数会得到一个泛型函数类型的值。
- SIL function类型可能声明为noescape
  - 函数声明的参数列表中，包含有noescape类型的function类型入参的话，该参数在降级时会被标注为@convention(thin) 或者 @callee_guaranteed. 这种情况属于unowned context,即context的生命周期必须保证能独立处理好.

```
FYI
    function类型作为入参的 escape特性默认值，对于Swift：
        <= 3.0  @escape as default
        >= 4.0 @nonescape as default

        ref：  https://github.com/apple/swift-evolution/blob/master/proposals/01
```

- SIL function类型可以声明对于context值使用何种调用约定来进行处理：
  - 对于@convention(thin)，表示函数没有context的值。这种类型也可声明为@noescape,即如果有context值，传递了也不会有额外的效果。
  - 对于@callee_guaranteed,表示context的值作为直传(direct)参数处理，且使用@convention(thick)的方式处理。若同时函数类型声明为@noescape，那么context的值是unowned;反之（@escaping），则context的值为guaranteed。
  - 对于@callee_owned，表示context的值为一个owned直传参数。且使用@convention(thick)方式处理。注意它并且与@noescape互斥，不可同时使用。
  - 对于@convention(block),表示context的值为unowned直传参数。
  - 其他的函数类型的规范会在Properties of Types和 Calling Convention部分讨论。

- SIL function类型声明了参数的调用约定。参数列表使用无标签元组(unlabeled tuple)的数据结构；tuple的每个元素必须是合法的SIL类型，并可以可选地添加以下某一种属性进行修饰：
  - 直传参数的值类型用T表示，间接参数的值类型用*T表示。
  - @in修饰间接参数。地址必须指向一个已初始化的对象；函数负责销毁所指对象。
  - @inout修饰间接参数。地址必须指向一个已初始化的对象。需保证在函数调用返回前该地址保持被初始化的状态。函数可以修改指针所指

（pointee），且可以弱假设不会发生对该实参进行别名化的读写（aliasing reads）。这要求该实参的值保持有效的状态，以便有序的别名违例(aliasing violations)不会影响内存安全。由此可以进行优化，如：local load/store propagation；引入和消除临时拷贝；把@inout参数提升为一个@owned直传参数和一个返回结果的方案。但是不允许优化导致该内存处于非初始化的状态。

```
FYI
  Pointer Aliasing
  多个指针的类型和所指向的地址是相同的，难以区分的情况，可能会引起问题。
  https://en.wikipedia.org/wiki/Pointer_aliasing
```

- @inout_aliasable修饰间接参数。地址指向已初始化的对象。需保证在函数返回前该地址始终是已初始化的。函数可以修改指针所指，同时必须假定参数的别名也会修改指针所指。可以假设这些别名的类型是正确的，且访问操作也是有序的。而不正确的类型（ill-typed）访问和数据竞争的结果是未定义的。
- @owned修饰owned直传参数。
- @guaranteed修饰guaranteed的直传参数。
- @in_guaranteed修饰间接参数。地址指向已初始化的对象。确保调用者和被调者都不修改指针所指，但被调用者（callee）可以读取。
- @in_constant修饰间接参数。地址指向已初始化的对象。函数中该值read-only。
- 此外，都是unowned直接参数。

- SIL function类型声明了调用结果（result）的约定规范。结果列表会写成unlabeled tuple；tuple的每个元素必须是合法的SIL类型，并可以可选地用下列属性进行修饰。有直接和间接结果两种（Indirect and direct results may be interleaved）.

```
FYI
原文中的result：  包括了 返回值 和 inout类型的参数
```

间接结果会出现在函数的entry block的隐式参数中，以及在apply和try_apply指令的参数中，用*T来表示。这些参数的顺序和在结果列表中的顺序一致，并且总是位于其他参数之前。

直接结果的类型表示为T。如果只有一个直接结果，那么返回结果类型就是这个值的类型；此外，返回类型是一个tuple类型的结果列表，tuple包含了所有直接返回结果。返回类型同时也是return指令、apply指令的操作数类型，以及try_apply指令的普通结果（normal result）的类型。

- @out，修饰间接结果。地址必须是未初始化的对象；函数负责初始化内存为一个有效的值。除非发生异常，使用throw指令退出，或者函数使用非Swift的调用约定。
- @owned，修饰owned的直接结果。
- @autoreleased，autoreleased的直接结果。必须是唯一的直接结果。
- 其他，unowned直接结果。

trivial类型的直传参数或结果总是unowned。

owned直接参数或者结果会把ownership转移给接受者，由其负责销毁。这意味着在值进行传递的时候，会引用计数会+1。

unowned的直接参数和结果，在传递的当下是有效的。接受者不用担心竞争条件会立即销毁该值，但还是应该先copy(通过strong_retain指令)再使用.

guaranteed的直接参数和unowned直接参数类似，不同在于调用者需要保证该值在调用过程中始终有效。由此可知，任何在被调函数中出现的strong_retain, strong_release指令对都可以优化消除。

autoreleased返回值的类型必须是支持retain的指针表示。autoreleased的结果在传递的时候不会在名义上对引用计数+1，但在运行时会安全地进行+1处理。这些处理要求精确的代码布局控制（Code-layout Control）。在SIL Pattern的表示上，autorelease的表示和owned表示看起来差不多，并且在SIL降级到LLVM IR的阶段，会在这两层的代码中插入额外的运行时指令。 对具有autoreleased结果的函数进行autoreleased调用(apply)，其结果在传递过程中引用计数会+1。 对定义不具有autoreleased结果的函数进行autoreleased调用(apply)，其结果引会被调用者进行强引用。 对具有autoreleased结果的函数进行non-autoreleased调用(apply)，其结果被被调者（callee）进行autorelease处理。

- SIL 函数类型可以接受一个可选的Error结果，写作@Error。Error结果总是具有隐式的@Owned属性。仅适用于native call规范。
  - 具有Error结果的函数必须使用try_apply指令进行调用。除非编译器可以证明函数实际不会throw，那么可以使用apply [nothrow]。
  - 用return指令返回普通结果。用throw返回error结果。
  - 对formal function类型降级的时候，throws注解会转化成更加具体的Error传播形式
    - 对于native Swift函数，throws会转换成一个error结果
    - 其他情况，会基于导入的API，将throws转换成显式的错误处理机制。导入器只会导入throws注解的非原生的方法和类型。如果可能的话，会自动这样处理。
- SIL函数类型提供了一种模式化签名(pattern signature)和模式替换（pattern substitution），通过特定的泛型抽象模式来表达该类型的值。这两者需要同时提供。如果使用了模式化签名，那么对应部分的类型（参数、结果、yields）都必须用泛型参数的签名方法来表示。另一方面，模式替换则应该用总体泛型签名的泛型参数来表达，或者使用闭包化的泛型上下文。
  - 模式签名在@substituted属性之后，它必须是函数类型前的最后的一个属性。
  - 模式替换跟在函数类型之后，在关键词 for 之后。
  - 如下例：

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

该类型的值的低级表示可能和它在替换过程中的版本不匹配。如下例：

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

convert_function指令可以对函数值在最外层的模式替换中的差异进行调整。注意，这只作用于最外层，而不会对嵌套的情况进行处理。比如，对于上面例子中两个类型，如果一个函数将前者作为参数类型，是不能直接转成后者的类型的。要完成这样的转换，要借助于thunk函数。

对函数类型的模式签名进行类型替换只会对其类型占位符进行替换；构件（Component）类型会保留原有的结构。

- 在实现上，SIL函数类型可以携带泛型签名的替换信息。这可以给泛型类型的应用和实现带来一些方便，但不算做SIL语言的正式标准。照理说泛型的值不应该携带这些类型。这样的类型仿佛是替换了底层的函数类型，其表现就像是非泛型类型。

Async Functions

SIL函数包括了@async类型。 @sysnc函数会运行内部的异步任务，并且可以显式地指定在一些代码点挂起程序执行。@async函数只能被其他@async函数调起，或者通过apply和try_apply指令来调用（或者如果是协程的话，可以通过begin_apply指令调起）

在Swift中，withUnsafeContinuation原语（primitive）用来实现执行的挂起点。在SIL中，@async函数使用get_async_continuation[_addr]指令和await_async_continuation指令来表示该抽象。get_async_continuation[_addr]可以访问一个continuation value，在协程被挂起之后，可以使用后者来恢复执行。该resulting continuation value会被传入一个completion handler,并注册到一个event loop里，或者有其他别的机制对其进行调度。 Continuation上的操作可以在async函数恢复执行的时候，将一个值或者一个error传回到async函数中。await_async_continuation指令会挂起协程的执行流，等待continuation将其唤醒恢复执行。

如下是withUnsafeContinuation在Swift中的使用：

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

可能会降级为如下的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
}
```

必须保证每个continuation value只对它对应的async coroutine使用一次。如果超过一次，则程序行为是未知的。另一方面看，如果没有成功恢复其执行，那么异步任务会卡在挂起的状态，相应的所属资源和内存都会泄漏。

Coroutine Types

协程(Coroutine)是指一个函数可以把它自己挂起，并把执行流的控制权交给调用者，而不用终止当次函数调用。因此，它可以不遵循严格的栈规则。SIL协程的控制流和其调用者紧密结合在一起。在yield点，可以以一种结构化的方式将值在caller和callee间进行双向传递。 Swift中的Generalized accessor和generator需要符合如下描述： 一个 read 和 modify的 accessor coroutine会映射到一个值，并且可以通过yield暂时地把该值的ownership交出来给caller。当coroutine恢复时，再把值的ownership收回。这样就使得coroutine可以清理相关资源，并对外部修改该值的行为作出相应的处理。 Generator的情况类似，它是流式yield值，但一次只yield一个值，同样临时交出值的ownership。 调用者控制流和这些coroutine的紧密耦合使得外部可以从coroutine借用（borrow）值。对于通常的普通函数而言，函数返回会转移返回值的ownership，因为在函数返回后其上下文就已经不存在了，也就不能继续维护返回值的ownership。

```
FYI  理论上，Coroutine可以把控制权交给任何人。
```

为了支持以上的这些概念，SIL支持两种协程：@yield_many 和 @yield_once。 将该属性写在函数类型的前面，以标志这个函数是一个协程类型。 @yield_many 和 @yield_once coroutine和@async关键词是兼容的。(您可以注意到但在SIL中@async函数本省不会显式地作为一种coroutine模型，只是在async在降级的时候使用coroutine作为它的实现方式。)

协程类型可以声明任意数量的yield的值。也就是说，yield给出的值都要声明在函数原型中。具体是写在函数类型的结果列表中，@yields作为前缀关键词。一个被yield的值会包含一个调用属性，该属性包含了一些调用参数，看起来就像从yield处调出，处理完又掉回该处。

目前，协程不会产生普通的结果（normal result）.

协程函数可以想普通的函数一样有很多种使用方式。但不能使用标准的apply 或者try_apply指令来调用。对于non-throwing 且 yield-once的协程，使用begin_apply指令调用。目前对于 throwing的协程 或者 yield-many的协程，暂时不支持调用。

协程可能包括特殊的 yield 和 unwind 指令。

@yield_many协程可以yield调出任意多次。@yield_once在函数返回前只能yield一次，但它可以在返回前throw错误。

Properties of Types

根据ABI稳定性和泛型约束，把SIL类型分类成了以下几组：

- loadable类型, 指可以完全暴露地用SIL进行具体表示的类型
  - 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

loadable类型的值可以对其本身或独立的子结构进行load和store。因此：

- loadable类型的值可以load进SIL SSA的值中，并且store操作不需要运行额外的用户编码，需注意编译器会自动生产ARC代码。
- 对loadable类型的值进行比特序拷贝，就可以获得一个初始化好的拷贝值

loadable的聚合类型包括了loadable的struct和tuple。

trivial类型是具有trivial 值语义的loadable类型。而且trivial 的load和store操作不需要ARC，其值也不需要销毁。

- Runtime-sized types,是指在编译器无法获得静态尺寸的类型，包括：
  - 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,是指不能load或以其他方式作为SSA值使用的类型
  - Runtime-sized types
  - Non-class protocol types
  - @weak types
  - 不满足loadable的要求的类型。比如某些值和内存位置相关联，在copy和move的时候需要运行一些用户定义的编码。最常见的情况是，这些值注册在全局作用域的数据结构上，或者其包含了指向自身的指针。比如：
    - @weak标记的弱引用值的地址被注册在一个全局表中。当该@weak值被copy和move到新的地址时，全局注册表也需要更新。
    - 堆上不支持COW(CopyOnWrite)的集合类型对象，（就像C++的std::vector），当集合被copy的时候，集合元素也需要一起被copy到新的内存地址。
    - 不支持COW的String类型，但通过持有指向自己的指针来实现一些优化的情况（就像C++里的std::string）。当String值被copy和move之后，该内部指针也需要更新。

address-only类型的值必须保证始终在内存里，SIL要使用只能通过内存地址引用。该类型的地址不可以进行load和store操作。SIL提供了一些专门的指令，间接地操作address-only类型，比如copy_addr和 destroy_addr。

另外，还有一些可以被分类的类型：

- 堆对象引用类型：其表示包含一个强引用计数的指针。包括：所有的class类型，Builtin.NativeObject，AnyObject，遵循class protocol的archetype。
- 引用类型在低级表示中可以包含额外的全局指针，和一个强引用计数指针。该类型包括所有的堆对象引用类型，thick函数类型，以及遵循class protocol的protocol的构件（composition）类型。所有引用类型都可以被retain和release，也可以拥有ownership语义的表示。
- 具有retainable指针表示的类型可以兼容ObjC的id类型。在运行时该值可能为nil。这包括了classes, class metatypes, block functions， class-bounded existentials with only Objective-C-compatible protocol constraints，还有用Optional或者ImplicitlyUnwrappedOptional对以上类型进行包装的类型。具有retainable的指针表示的类型可以使用@autoreleased的规范进行return。

SILGen不能保证将Swift函数类型一一映射到SIL函数类型。函数类型的转换就是给其编码一些额外的属性：

- 关于调用约定的语义，通过以下属性表示：

```
@convention(convention)
```

该属性和语言层面的@convention有点像，但在SIL层扩展了多个额外的版本：

- @convention(thin)，thin函数：使用swift调用约定，并且不会额外传递"self"或"context"参数
- @convention(thick)，thick函数：使用swift调用约定，并且传递一个引用计数的context对象。context捕获上下文信息或者持有函数的状态。该属性具有@callee_owned 或者 @callee_guaranteed的Ownership语义
- @convention(block)，ObjC兼容的block。兼容ObjC的ID类型，其对象包含了block的调用函数，其使用C的调用约定。
- @convention(c)，C函数：不携带context，使用C函数调用约定：使用C调用约定，在SIL层会携带一个self参数，在实现层面，映射为self和_cmd参数。
- @convention(objc_method)，使用ObjC方法实现。
- @convention(method)，Swift实例方法实现。使用Swift调用约定，携带特殊的self参数。
- @convention(witness_method)，Swift协议方法实现。通过这样一种函数的多态约定方式，来确保该协议的所有实现者都可以通过多态的方式使用该函数。

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中以$作为值的前缀。值的命名使用字母和数字来作为唯一标示，该id可以引用指令或者basic block的实参。 也可能是'undef',如字面义。

和LLVM IR不同，把value作为操作数的SIL指令只能接受value作为操作数。对于字面量常量，函数，全局变量和其他实体进行操作则需要专门的指令进行操组，如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函数使用关键字sil来定义。SIL函数名以@符号开始，后面接字母数字构成的id。这个函数名也是LLVM IR中该函数的名字，但通常会使用原始的swift声明进行mangle编码。 这个sil句法声明了函数的名字，SIL的类型并且定义了花括号里的函数体。声明的类型必须是函数类型，当然也有可能涉及泛型。

Function Attributes

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

该函数已经是canonical SIL，即使这个模块目前还只是在Raw SIL的状态。

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

该函数遵循 OSSA(ownership SSA)的形式。

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

透明函数(transparent function)总是内联的，并且内联后不会保留源码信息。

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

该函数是编译器生成的thunk函数。

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

该函数可以在运行时替换成不同的实现。优化动作不应该对该类型的函数作任何假设，即便可以获得函数体的SIL源码也不行。

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

以上方式可以指定用哪个函数替换那个目标函数。

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

该函数是class的designated initializer，在该函数体内部会alloc该class的静态类型。

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

该函数是闭包的thunk函数，其并不会escaping。

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

以上的属性，表达的语义如下：

- 在第一次调用之前，任何时候都可能会有副作用
- 对同一个global_init函数的调用具有相同的副作用
- 观测initializer副作用的任意操作都必须先于对该initializer的调用

当前是成立的，如果该函数是访问全局变量时候lazy生成的一个寻址器（addressor）的话。 注意，初始化函数本身并不需要此属性。该函数是private的，并且只会从寻址器内部被调用。

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

该函数是 lazy属性的getter函数，且该属性后端存储的类型为Optional。 对应的getter方法包含了一个top-level的switch_enum（或 switch_enum_addr），后者会检查是否这个lazy属性已经被计算出来。

lazy属性的getter被首次调用之后，则可保证该属性已经被计算出来，且后续调用总是会执行top-level switch_enum的一些case。

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

跨模块引用该函数需要使用weak链接。

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

指定该函数最低的系统可用版本。

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

该函数绝不内联。

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

该函数总是内联，即使使用Onone编译参数。

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

函数优化编译属性，该属性的局部优先级高于编译命令的优化参数。

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

指定函数的内存效果。

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

函数指定的high-level语义。优化器可以在内联之前，使用这些信息进行high-level的优化。 比如，Array操作可以注解一些语义的属性，使得优化器可以进行冗余的边界检查消除优化和其他类似的优化。

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

指定哪些类型指定的代码应该被生成。

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

The clang node owner.

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
```

一个函数的body包含至少一个basic block,bb的数量是由函数的CFG(control-flow-graph)的节点数决定的。 每个BB包含至少一条指令，并且以一条terminator指令结束。 函数入口总是其函数体的第一个BB。

SIL中，BB里包含了参数，这类似于LLVM的phi节点。BB的实参由其分支的前序bb进行绑定。

```
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
}
```

如果当前BB没有前序的BB，那么它的实参由调用者绑定：

```
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 : $()
}
```

如果该函数应用了Ownership SSA的规范，那么参数会显示地添加一些规范的注解信息，用来描述参数的Ownership语义。

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

注意，以上代码里的第一个参数%x拥有一个隐式的@none ownership语义，这是由于Int属于trivial type，默认是none，是符合规范的。

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]+
```

每条指令都可有一个debug的位置信息和一个SIL scope引用信息在最后。Debug的位置信息包括了文件名，行号，列号。如果debug位置信息是自动输出的，那么它的默认位置是SIL源文件。SIL Scope信息描述了生成SIL指令的Swift表达式最初在词法scope结构内的位置。SIL scopes同时保留了inline的信息.

Declaration References

```
sil-decl-ref ::= '#' sil-identifier ('.' sil-identifier)* sil-decl-subref?
sil-decl-subref ::= '!' sil-decl-subref-part ('.' sil-decl-lang)? ('.' sil-dec
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]+
```

有些SIL指令需要直接引用一些Swift声明。这些引用以#符号开头，后接限定的Swift声明的名称。有些Swift声明被拆解成不同的实体，分布在不同的SIL level上。它们具有以下特征：限定名称以！开头，并且用一到多个.符号进行实体分割：

- 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 Swift 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'
```

链接描述符控制了不同SIL模块中的对象如何链接，也就是如何当作同一个对象。

一个链接如果有 external的后缀，那么它是external的。 一个对象的链接如果不是external的话，那么它在当前模块内必须有定义。

所有的函数，全局变量，以及vitness table 都有链接。 定义的默认链接描述定义为public；而声明的默认值为public_external。（有可能以上默认值逐渐会对应改为hidden和hidden_external。）

对于全局变量的外部链接意味着该变量不是一个定义。一个变量如果缺少显式的链接描述符则会被推断为一个定义（因此会获得定义的默认链接描述，public）。

Definition of the linked relation

如果对象如果互相可见，并且具有相同的名字，那么就会被链接起来。

- 具有public 或 public_external链接描述的对象总是可见的
- 具有hidden 或 hidden_external，或shared 链接描述的对象只对模块内的对象可见
- 具有private 链接描述的对象只对模块内的对象可见

请注意，链接关系是等价关系：它是自反的，对称的和可传递的。

Requirements on linked objects

要使两个对象链接，他们必须具有相同的类型。

要使两个对象链接，他们必须具有相同的链接描述，除了：

- public对象可以链接到public_external对象上
- hidden对象可以链接到hidden_external对象上

要使两个对象链接，则两者至多有一个是定义，除非：

- 两者都具有shared属性
- 至少其一具有external属性

如果两个对象都是定义，且链接在一起了，那么这两个定义在语义上必须是等价的。这种等价可能仅存在于定义良好的代码的用户可见语义上；但不应该认为其保证链接定义在操作上也完全等效。 例如，一个函数的定义可能会从地址参数中复制一个值，而另一个函数分析证明了并不需要该值。

如果一个对象被使用了，那么它必须被链接到一个with non-external 的定义上。

Public non-ABI linkage

non-abi链接是比较特别的链接，它仅提供给序列化的SIL中的定义使用的，它不会在object文件中定义可见的符号。

具有non-abi链接描述的定义有点像shared链接，区别在于它必须被序列化，即使没有被当前模块的任何位置所引用。例如，它可能会被当作无用函数消除优化（dead function elimination）的根。

当一个non-abi的定义被反序列化后，它将具有shared_external链接描述。

并没有non*abi_external这种链接描述。相反，当需要引用同一Swift模块但不同转换单元中定义的non_abi声明时，必须使用hidden*external链接描述。

Summary

- public定义在程序中是唯一的，且到处都可见。在LLVM IR中，他们被输出为external链接性以及default的可见性。
- hidden定义在当前Swift模块中唯一且可见。在LLVM IR中，他们被输出为external链接性以及hidden的可见性。
- private定义在当前Swift模块中唯一且可见。在LLVM IR中，他们被输出为private链接性。
- hidden定义在当前Swift模块中唯一且可见。在LLVM IR中，他们被输出为external链接性以及hidden的可见性。
- shared定义在当前Swift模块中可见。仅可与其他等价的shared定义链接；因此，只有确实被用到了才需要被输出（emit）。在LLVM IR中，他们被输出为linkonce_odr链接性以及hidden的可见性。
- public_external和hidden_external对对象始终在其他位置具有可见的定义。如果此对象仍有定义，那仅是出于优化或分析的目的。在LLVM IR中，其声明具有external链接性，并且定义（实际会输出为定义）具有available_externally链接性。

VTables

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

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

SIL用 class_method, super_method, objc_method, and objc_super_method 指令来表示class方法的动态派发。

每个class类型都会有声明一个sil_vtable,可以在其中查找class_method 和 super_method。Vtable包含了class方法（包括继承的）到具体SIL函数实现的映射：

```
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
}
```

可以注意到派生类C的A.bar是从派生类B继承而来的，因此vtable传递遵循最近可见性原则。Swift AST维护了这些覆写关系，从而可以查找派生类的覆写方法。

如果SIL函数是一个thunk函数的话，那么会根据函数名在原函数实现的链接描述之前。

Witness Tables

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

SIL把泛型动态派发所需要的信息编码进WitnessTable。在生成二进制代码的时候，用这些信息来生成运行时方法派发表。这些信息也会被用于SIL优化，例如泛型函数特化。协议的每个显式声明的遵循（conformance）都会生成一个WitnessTable。泛型类型的所有实例会共享同一个泛型WitnessTable。派生类会从基类继承WitnessTable。

```
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
```

Protocol conformance是某具体类型对协议的具体遵循，它具有唯一标示性，并用来作为WitnessTable的键。

- 一个normal protocol conformance描述了其(潜在的未绑定泛型unbound generic)类型，它所遵循的协议，以及其类型，扩展声明该conformance所属的模块。从而源码里的protocol conformance声明能一一对应上。
- 如果基类遵循了某协议，那么派生类将通过一个inherited protocol conformance继承，它只是简单地引用了基类的protocol conformance。
- 如果一个泛型的实例遵循了某协议，那么会得到一个specialized conformance，它可以给normal conformance绑定泛型参数的。

```
// FYI
如下为 witness table for unbound generic conformance
sil_witness_table hidden <M> C<M>: Greeting module wt3 {
  associated_type T: M
  method #Greeting.sayHi!1: <Self where Self : Greeting> (Self) -> (Self.T) ->
  method #Greeting.sayBye!1: <Self where Self : Greeting> (Self) -> (Self.T) -
}
```

Witness tables只和normal conformance直接关联。Inherited 和 specialized conformances都是间接地引用对应的normal conformance的witness table。

```
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-conforman
```

Witness tables 由以下的入口(entry)构成:

- Base protocol entries,通过提供对protocol conformance的引用，来实现witnessed protocol的继承关系。
- Method entries，提供了协议里面声明的方法到对应方法实现的映射。witnessed protocol里的required方法必须对应到一个method entry。
- Associated type entries，提供了协议里声明的associatedtype到具体的witness type的映射。需要注意的是，witness type是Swift语言层面的类型，而不是SIL类型的。witnessed protocol里的required associatedtype必须对应到一个Associated type entry。
- Associated type protocol entries，提供了associatedtype要求的protocol 对应到 protocol conformance的映射。

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在default witness table里编码了用弹性实现的协议条款（requirements）。 如果满足以下条件的话，我们就说协议里的条款(requirement)有一个弹性的默认实现：

- 该requirement具有一个默认实现
- 该requirement要么是协议中的最后一个，要么其所有后续的requirement都具有弹性的默认实现

具有默认实现的requirements的集合存储在protocol的元数据中。

```
Requirements 协议条款
就是protocol中声明的约束，包括可选部分和必选部分。
具体还可分为，方法，属性等。
```

minimum witness table size就是不包括任何弹性默认实现的witness table的尺寸。

default witness table的实际尺寸等于 最小尺寸 + 默认requirements的尺寸，但不会超过最大尺寸。

在加载的时候，如果运行时发现witness table的尺寸达不到最大尺寸（也就是说缺少部分requirement），那么会拷贝一份新的witness table，并从default witness把缺失的部分填充拷贝填充进来。这就保证了调用者在witness table中总是可以找到预期数量的requirement。并且framework的作者可以添加新的requirements，而不用打断客户端的代码执行，前提是新的requirements也有弹性的默认实现。

Default witness table是由协议自身来标示的。只有具有public可见性的协议需要default witness table。private和internal协议不会被模块外可见，因此添加新的requirement不会有弹性的问题。

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

Default witness tables目前只包含一种entry：

```
- Method entries，把协议的方法requirement映射到一个SIL函数，该函数实现了对所有witness类
```

Global Variables

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

SIL可以表示全局变量，通过alloc_global, global_addr 和 global_value指令进行访问。

全局变量可以有一个静态初始化器（initializer），但要求它的初始值由字面量(literals)组成。初始化方法用字面量列表和聚合指令来表示，后者的最后一条指令是静态initializer的顶层值（top-level value）：

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

如果一个全局变量没有静态initializer，那么必须在首次访问之前使用alloc_global指令来申请存储。一旦全局存储初始化了，接下来使用global_addr指令来投影(project)该值。

如果静态initializer的最后一条指令是object指令，那么全局变量是一个静态初始化的对象。这种情况下，该全局变量不能作为左值，也就是说对这个对象的引用是不能修改的。于是，这个变量只能使用global_value访问，而不能通过global_addr访问。

Differentiability Witnesses

```
FYI 可微分witness，特定场景才用得到，不翻译了。
```

0x3 Dataflow Errors

数据流错误出现在raw SIL中。Swift语义上定义了下列条件为error，因此他们必须被诊断pass检测出来，不能让他们出现在canonical SIL中。

Definitive Initialization

Swift要求所有局部变量在用之前都进行初始化。 所有struct、enum、class等类型的实例变量都必须先初始化再使用，由构造器进行初始化。

Unreachable Control Flow

在raw SIL中会输出unreachable terminator来标记错误的控制流，例如non-Void返回值的函数返回了一个值，或者switch没有覆盖所有case等。DCE(dead code elimination，无用代码消除)优化器会把这些不可及的bb代码块给消除掉；另外对于返回uninhabited类型的函数的apply指令，如果是unreachable也可以消除。 在DCE阶段留下来的不可及指令，但没有立即被另一个non-return application处理的话也会认为是一个数据流错误。

```
Uninhabited Types
Never is an uninhabited type, which means that it has no values. Or to put it

https://nshipster.com/never/
```

0x4 Ownership SSA

SILFunction如果被标记为[OSSA], 则表示该函数遵循Ownership SSA的形式。OSSA是实参版本的SSA，通过给值操作边界上添加Ownship语义信息，从而确保实参的Ownership不变。 所有的SIL的值都被静态地赋予了一个Ownership语义。所有对SIL值进行操作的SIL操作指令被分为两类：

- “normal uses”， 要求操作数，也就是操作的SIL值是有效（live）的
- “consuming uses”，消费该操作数，结束这个SIL值的生命周期，之后不可再使用该SIL值。

因此，所有的非消费类的操作指令都必须在“consuming uses”之前进行。 SIL值在所有可及的（见程序可及性）程序执行路径下，必须且仅被消费一次。 由此可以检查SIL值的内存泄漏和过度释放，野指针等问题。

看一个例子，注意定义和使用是成对出现的，并且用注释标明了：

```
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
}
```

可以注意到SIL中的每个值都是成对使用的（通过normal use和 consuming use）。如果违反了OSSA规则，会引发一个SILVerifier的错误，从而定位泄漏和野指针问题。

上面例子中只使用到了“Owned”语义，事实上，在SIL中，支持共四种Ownership语义：

- None
  - 表示该值不需要进行内存管理，不用遵循OSSA的不变性的要求。例如：
    - trivial values (e.x.: Int, Float),
    - non-payloaded cases of non-trivial enums (e.x.: Optional.none)
    - all address types
- Owned
  - 该值具有独立的Ownership，不依附其他外部值。
  - 在可及的程序路径下只能销毁（通过destroy_value）或消费（通过apply,cast,store等对值重新绑定）一次。

- Guaranteed
  - 该值的生命周期不独立，依附于”Base Value“——其他owned或Guaranteed值。
  - 必须Guaranteed它的base value销毁/消费前，该值的有效性
  - begin_borrow指令开始，end_borrow结束
  - base value需要能静态地确保其值的有效性。
- Unowned
  - 值仅当前有效。使用的话，必须copy成一个Owned或者Guaranteed的新值。
  - 用于OC unsafe unowned 入参传递
  - 也用于，把trivial值按位转成non-trivial类型的值。
  - unowned值不可被consume

Value Ownership Kind

Owned

Owned所有权描述的是“仅作移动”的值。 Owned值在所有可及的程序路径下，需要且只能被消费一次。 IR检查器把始终未consume的Owned值标记为Leak；并且把多次consume的值标记未use-after-frees。 可以通过“forwarding uses”来建模move操作。这包括了cast
和 transforming terminators(e.x.: switch_enum, checked_cast_br)。后者会转换输入值，在转化过程中consuming它，并生成一个转换好的owned的新值作为结果。

我们可以把每个owned SIL值当作“仅移动的值”，除非值被显式地进行copy_value。这样的话，ARC就可以只对特定的值进行语义上的操组，而且由于SILValue遵循SSA，因此ARC对象在声明生命周期中可以派生出多个独立的Owned SILValue，后者具有独立的生命周期和作用域，可以进行独立的检查。

例子如下：

```
// 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 : $OtherKla

  // 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>
}
```

可以留意一下(%1)是如何被消费、转发、并且映射到新的值 (%2, %3, %5)上，而后者也都有独立的Owned值寿命。

Guaranteed

Guaranteed值寿命依赖于“base value”。base value具有owned或者guaranteed属性。guaranteed值有效的前提是base value有效，这就要求在所有的范围下，base value寿命范围必须覆盖guaranteed值的。

这需要显式地使用 begin scope指令，如"begin_borrow","load_borrow"。并且需要成对地出现end scope指令： "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:
  ...
}
```

可以注意到，%0的销毁指令总是在%1之后，所以编译优化器是无法在%1销毁前缩短%0的生命周期的。

对于具有guaranteed所有权属性的值，遵循以下数据流规则：

> non-consuming的forwarding uses指令，输入值为base value，生成值为guaranteed的值。 此规则适用于递归的场景。

从而，新值依赖于原值的生命周期和对其对应的scope。 这么设计是为了这避免了生成过多幂等的scope（每个scope都需要一对指令生成）。

```
sil [ossa] @get_first_elt : $@convention(thin) (@guaranteed (String, String)) 
bb0(%0 : @guaranteed $(String, String)):
  // %1 is validated as if it was apart of %0 and does not need its own begin_
  %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

None类型的Ownership表示该值不需要进行内存管理，OSSA机制需要忽略的部分。

包括如下类别：

- Trivial type (Int, Float, Double)
- Non-payloaded non-trivial enums
- Address types

```
F.Y.I.
对于Non-payloaded non-trivial enums，和Swift对enum的实现有关。
对于没有payload的enum，最简化的实现版本类似C的enum，因此可以看作是一种特殊的Trivial Type
```

none ownership值被排除在OSSA之外，以昵称他们可以被当作普通的SSA来用，而不用担心破坏来OSSA 不变性的规则。但这并不意味着该代码就不违反其他的SIL规则（例如内存生命期不变性规则）。

```
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

适用于以下两个场景：

- ObjC函数调用规范中的入参。
  - 该规范要求调用者对入参先copy再使用。
  - SIL要求owned和guaranteed值作为入参传递前必须先copy。
  - 译者：如果同时遵循两个规范，那么需要在传参的前后copy两次，显然不合理。
  - 目前SIL还没有对这个场景专门设计一个机制处理，所以先用Unowned
- 从None Ownership的trivial type value 转过来的 non-trivial type value
  - 例如把一个trivial的unsafe指针值转成一个Class*的值。我们不能保证这个被指class对象的生命周期，所以我们必须先把这个值copy一下再用。

Forwarding Uses

SIL的部分子集指令根据操作数的值所有权类型定义其结果的值所有权类型。我们称这种指令为“forwarding指令”，任何使用该用户指令的用例我们称为“forwarding use”。这种推断通常发生在指令构造时，因此：

- 由于大多数情况下ownership是体现在指令中的，因此若通过程序操作forwarding指令，那么必须手动更新ownership forwarding。
- SIL文本并不能显式地表示出forwarding指令的所有权。相应的，指令的ownership是通常是通过解析后的操作元推断出来的。由于SILVerifier会在解析之后检查文本化SIL，因此你可以认为ownership的约束已经被正确地推断出来了。

根据构造操作数的值ownership类型不同以及返回类型不同，Forwarding的语义会有轻微的不同。 如下讨论：

- 如果操作数是@owned,forwarding指令会结束原操作数的生命，并生成一个新的值：对non-trivially类型生成@owned，对于trivially类型，则生成@owned。例如：用来表达类型转化的语义。

```
sil @unsafelyCastToSubClass : $@convention(thin) (@owned Klass) -> @owned SubK
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
}
```

- 如果操作数是@guaranteed,那么forwaring指令针对non-trivially类型值会生成@guaranteed值，对于trivially类型值会生成@none值。对于non-trivially的例子，指令会隐式地对输入值开始一个新的begin borrow。由于这个borrow scope是隐式的，因此我们会像检查操作数一样检查结果的使用（支持递归）。这意味着对于任何guaranteed forwarded结果，我们不会看到对应的end_borrow指令。实际上end_borrow总是通过引入borrowed值的指令达成。一个guaranteed forwarding 指令的例子是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) -> @owne
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
}
```

- 如果操作数是@guaranteed,那么结果只必定是@none。

- 如果操作数是@unowned,那么结果只必定是@unowned。会像检查@unowned值一样检查该值，也就是确保先copy再使用。

这里的另一个问题是，即使绝大多数forwarding指令forwarding所有类型的ownership，但一般的情况并非如此。至于为什么，我们可以比较一下struct_extract（不forward @owned ownership）和unchecked_enum_data(forward所有ownership类型)。

不同的原因在于，struct_extract本质上只能提取较大对象的单个字段，这意味着该指令只能表示使用某个值的子字段（sub-field），而不是一次使用整个值。这违反了我们的约束，即，@owned值不能被部分consume：值要么完全live，要么完全dead。 另一方面，enum总是把元素装在一个tuple里来表示payload。这意味着使用unchecked_enum_data指令提取enum中的payload时可以consume这个那个enum和payload。

若想使用struct_extract来consume struct的话，我们可以使用destruct_struct指令，该指令一次consume整个struct，然后将struct的各个组成部分返还给用户。

```
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 valu
  (%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
}
```

0x5 Runtime Failure

某些操作（例如失败的无条件检查转换（checked conversions）或内置的Builtin.trap编译器）会导致运行时失败，从而无条件地终止当前actor。如果可以证明将发生或确实发生了运行时failure，则可以对运行时failure进行重新排序，只要它们相对于actor或整个程序的外部操作保持良好的顺序即可。例如，启用对int算术的溢出检查后，一个简单的for循环会从一个或多个数组中读取输入并将输出写入到另一个数组中，这些数组都是当前actor本地的变量，这可能会导致更新操作中的运行时失败：

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

允许将溢出检查和相关的运行时failure从循环本身中提升出来，并在进入循环之前检查循环的边界，只要循环体在当前actor之外没有可观察到的外部作用即可。

0x6 Undefined Behavior

不正确地使用某些操作会导致未定义行为（Undefined Behavior）。例如涉及Builtin.RawPointer的无效的未经检查的类型强转，或使用在LLVM level使用具有未定义行为的低至LLVM指令的编译器内置函数。具有未定义行为的SIL程序是无意义的，就像C中的未定义行为一样，并且没有可预测的语义。正确的Swift程序使用正确的标准库输出的有效SIL不应触发未定义行为，但不是所有情况都可以在SIL level得到诊断或验证。

0x7 Calling Convention

Swift Calling Convention @convention(swift)

Swift调用约定（Calling Convention）是Swift原生函数使用的。

函数输入类型里的tuple会被递归地解构成分开的实参。该过程发生在basicblock的入口处，或者在调用者使用apply指令的时候:

```
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)) -> ()
}
```

如果函数调用的输入输出类型都是trivial类型的话，那只要简单地对参数传值就可以了。如下函数：

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

foo(x, y)
```

在SIL中调用：

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

Reference Counts

注意⚠️ 本节的内容仅针对经验法则。实参的实际行为是由实参的约定属性（例如，@ owned）定义的，而不是调用约定本身。

引用类型的实参在传递的时候会被retain +1，由callee进行consume。 引用类型的返回值在return的时候会+1，由caller进行consume。 具有一些引用类型所构成的值类型，会和其成员引用类型一起retain和release。

```
FYI consumed parameters
A function or method parameter of retainable object pointer type may be marked
ref: https://clang.llvm.org/docs/AutomaticReferenceCounting.html#consumed-para
```

```
class A {}

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

bar(x)
```

SIL调用：

```
%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
```

当callee apply一个thunk函数时，该函数作为值也会被consume，即retain +1。

```
FYI
apply/application
在当前语境是FP中的概念，即实参绑定到函数上下文并调用。
```

Address-Only Types

对于Address-Only类型的实参，caller会alloc生产一个拷贝副本并且把副本的地址传递给callee。callee接管副本的ownership，并负责对其销毁或consume，而caller仍然负责dealloc对应的内存。 对于Address-Only的返回值，caller alloc一块未初始化的内存作为buffer，并且把该地址作为第一个参数传递个callee。而callee必须在返回前初始化这块buffer。如下：

```
 @API struct A {}

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

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

在SIL中是这样的：

```
%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
```

@bas的实现负责consume %x_arg，并且初始化%z。

Tuple类型会被析构，并忽略tuple类型是否是address-only-ness。析构的多个field会根据上述调用约定被独立地传递。如下：

```
@API struct A {}

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

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

在SIL里看起来是这样的：

```
%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

不定长参数以及tuple的元素会被打包成一个数组，并且作为一个单独的实参进行传递。如下：

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

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

其SIL如下：

```
%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实参通过地址传递 在到函数的入口处。callee不会接管引用内存的ownership。在函数入和退出时，其引用内存必须是已初始化的状态。 如果@inout实参引用一个fragile physical变量，那么这个实参是这个变量的地址。 如果@inout实参引用的是一个逻辑属性，那么这个实参是caller-owned 的可回写的buffer的地址。这时caller负责在调用函数之前属性的getter，并用getter的结果写入并初始化buffer；在return时从buffer中load出值，并用最终值调用setter来写回属性。

```
FYI  fragile physical 变量
In-Out 参数传递有两种方式：
  copy-in copy-out 和 call-by-referernce
  其中后者为优化，可省略copy过程
ref：  https://docs.swift.org/swift-book/ReferenceManual/Declarations.html
```

Swift代码如下：

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

对应SIL如下：

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

Swift Method Calling Convention @convention(method)

当前，方法调用约定与单个（freestanding）参数函数的调用约定相同。可以认为方法都柯里化（Curried ）了，最外层的是self参数，其他方法参数在内层。 因此，self参数在最后传递：

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

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

Witness Method Calling Convention @convention(witness_method)

Witness方法调用是指WitnessTable里的协议方法。它与方法调用约定相同，不同之处在于它对泛型类型参数的处理。对于non-witness方法，在machine层面关于传递类型形参的元数据的约定可能仅依赖于函数签名的静态部分，但因为witness必须支持对self类型的多态派发，因此传递与Self相关的元数据必须以尽可能抽象的方式。

C Calling Convention @convention(c)

在Swift的C模块导入器中，C类型总是被SIL映射到Swift的trivial类型。 SIL不关心平台ABI关于indirect return，寄存器与堆栈传递等要求。SIL中的C函数的传递和return总是传值，而忽略平台的调用约定。

因此目前SIL和Swift不能调用具有不定长参数的C函数。

Objective-C Calling Convention @convention(objc_method)

Reference Counts

Objective-C方法使用与ARC Objective-C相同的实参和返回值ownership规则。Selector以及Objective-C定义的ns_consumed，ns_returns_retained等属性也都支持。

apply @convention(block) 的值，并不会consume这个block。

Method Currying

OC方法的参数在SIL里也会柯里化，包括self参数，就像原生Swift方法一样。

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

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

在SIL IR层，self和_cmd作为方法调用前面的参数，都被抽离出来了，默认隐藏了。

GOSSARY

fragile physical变量 | | inout参数传递 callbyreference优化

0x8 Type Based Alias Analysis

SIL支持两种基于类型的别名分析（TBAA）: Class TBAA 和 类型化访问的(Typed Access) TBAA。

Class TBAA

对Class实例和堆上其他的对象的引用是通过指针来实现的，这和SIL中的address类型不同，后者是SIL类型的一等公民(first-class)，可以被捕获(captured)和别名化(aliased)。 Swift是一个内存安全、静态类型的语言，所以Class的别名必须符合以下约束：

- Builtin.NativeObject类型可以作为堆上任意的Swift Native object的别名，包括swift class实例，通过alloc_box创建的box类型，或者thick的函数闭包。但不能作为ObjC对象的别名。
- AnyObject 和 Builtin.BridgeObject 可以作为任意Class实例的别名，包括OC和Swift都可以，但不包括堆上非Class实例。
- 两值Class类型的关联性
  - 如果两个值都是是同一个Class类型，可以互为别名；
  - 如果一值类型为$B,一值类型为$D,$B、$D是继承关系，则可以别名化；
  - 如果两值的Class类型不是关联的，则不能别名化。包括了类型特化了的泛型Class，例如 $C < Int >, $C < Float >
- 由于局部代码没有全局的可见性，因此必须假设泛型和协议的占位类型(泛型对应archetype, 协议对应 typealias)可能是任意的class实例的别名。即使从局部代码来看，某个class可能没有遵循特定协议，但在模块外可能它的某个extension实现了该协议。对于泛型，同理。

如果违反以上规则，在swift代码中解引用别名的引用值，那么程序可能会出现未定义的行为。 举个例子，__SwiftNativeNS[Array|Dictionary|String] classes 其实是NS[Array|Dictionary|String] classes的别名，但他们静态关联的。但是由于Swift不会对Foundation Class实例的存储类属性(StoredProperty，区别与计算型)进行直接的访问，因此对他们的别名化不会导致问题。

Typed Access TBAA

对地址或引用的类型化访问(typed access)，定义如下：

- 对给定位置的内存进行类型化读写访问的任意指令，(e.x. load, store).
- 通过类型化投影操作获得指针的类型化偏移量的任意指令(e.x. ref_element_addr, tuple_element_addr).

除了个别例外，对没有绑定到对应类型的地址或引用进行类型化访问，程序行为是未知的。

由此，优化器可以推断，对于两个Address，如果找不到一个archetype进行替换，使得替换后的两个类型T1，T2能满足T1正好是T2的子对象（？？ 没理解 怀疑写错了）的类型，那么这两个Address也不能alias。 （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. ） 以上规则同样适用于从类型化投影（typed projection）获得的值的类型。

看下面的例子：

```
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
```

优化器判断%adr1 不能为 %adr2做别名，因为他们是从%ptr1(S1) 和 %ptr2(S2)里衍生出来的，S1和S2是不相关的类型。 然而在下面的例子里情况不同，%adr2是用一个类型转化得来的，而接下来的类型化操作都只和Element相关，%adr1对应S.slt: Element, %adr2对应$*Element, 因此他们是可能alias的。

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

typed TBAA也有例外，仅限于alias-introducing operations（别名引入操作）。这个机制有限地允许类型双关(type-punning).目前唯一的例外的场景是pointer_to_address指令的non-struct变种。 优化器必须防御性地判断所有的alias-introducing operations都是address-roots。 Address Root是指生成address的指令，这类指令先于任意类型投影，索引，转换指令之前执行。以下都是有效的address roots：

- Object allocation指令，其会生成一个address, 比如 alloc_stack, alloc_box指令
- Address类型的函数实参。该场景不被认为是alias-introducing operations。SIL优化器从任意的address类型实参生成新的实参是非法的。因为这要求优化器能保证新的函数实参不是alias-introducing address root，且在函数调用约定中能用很好地方式表示出来（address类型没有固定的表示方法）。
- pointer_to_address [strict]，对一个没有类型的指针进行严格的类型转化。pointer_to_address[strict]从alias-introducing operation的值生成address是非法的。目前，类型双关的address只能是用非strict模式的pointer_to_address指令，从不透明指针(Opaque Pointer)生成。

使用 unchecked_addr_cast 指令进行 Address-to-address的类型转换，像类型投影一样，可以透明地forward address root。

Address类型的BasicBlock参数可以保守地认为是aliasing-introducing operations。它们很罕见，但不重要，最终可能会被完全禁止。

有一些指针的生成是天生就存在的，不需要作为TBAA的alias-introducing例外情况来考虑。例如，Builtin.inttoptr 会生成一个 Builtin.RawPointer类型的指针，可以任意地别名。类似的，LLVM内置的Builtin.bitcast 和 Builtin.trunc|sext|zextBitCast也

不生成类型化的指针。 如果要对这些指针进行类型化的访问，必须先使用pointer_to_address指令进行转化。 而pointer_to_address是否是strict模式决定了是否可以别名。

内存可以被重新绑定到一个不相关的类型上。如果类型化访问的内存被绑定了具体类型，那么Address也可能为不关联的类型alias。因此，优化器也不能完全确定被访问的两个具有不相关类型的Address就不能alias。 例如，不能简单粗暴地消除指针比较，因为从这些指针派生的两个Address可能在不同的程序执行点上被当作不相关的类型进行访问。

0x9 Value Dependence

总的来说，分析器可以假定独立的值独立保证其值的有效性。 比如以下的例子，一个类方法返回一个类引用：

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

优化器可以把strong_release %1这一行移动到函数apply之后，因为编译器认为%2是独立管理的值，并且Swift一般是允许调整解构顺序的。

有一些指令的执行会创建操作数之间的依赖关系。 比如，如果获取一个结合类型的对象内部元素的指针，而这个集合对象在指针放阿文钱释放里，那么就会出现野指针。 因此，也就需要 Value-Dependence的概念。所以，对解构顺序进行优化重排的化，需要特别注意值之间的依赖关系。

在以下场景，我们可以说值%1依赖于值%0:

- 对于以下指令，%0是第一个操作数，%1是操作结果
  - 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
- mark_dependence指令，%1是操作结果，%0是任意操作数
- box_alloc指令，%1是address，%0是ref
- struct,enum,tuple指令，%1是结果，%0是操作数
- tuple_extract, struct_extract, unchecked_enum_data, select_enum, or select_enum_addr指令，%0是聚合对象，%1是提取出来的子对象
- select_value指令，%1是结果，%0是某个case（switch case）
- 对于一个BasicBlock，%1是形参，%0是实参
- 对于load指令，%0是地址操作数，%1是结果。然而如果%1一旦被managed，这种依赖关系就切断了，（因为%1可以独立管理了）。例如retain %1。
- 依赖的传递性。但是这个性质不能引用到struct、tuple、enum的不同子对象上。

另外，需要注意：

- 对内存的依赖关系进行追踪分析不是必要的；
- 对不透明机制的代码进行“透明的”依赖关系分析也不是必要的。比如一个方法返回一个执行类属性的unsafe pointer。

- 对函数内部SIL指令的依赖关系分析是必要的。
- 对于SILGen(用mark_dependence指令是否合理)和用户（是否合理地使用了unsafe语言提供的一些属性或者库的特性）是否生成了合适的依赖关系，需要特别注意。

只有特定的SIL Value可以携带值依赖关系的信息：

- SIL address types
- unmanaged pointer types:
  - @sil_unmanaged types
  - Builtin.RawPointer
- 包含像UnsafePointer类型元素的聚合容器；有可能形成递归依赖关系
- non-trivial types (但也可被独立managed)

依据以上规则，把pointer转成Int，就丢失了携带的依赖关系。但是如果只是为了读Int值的话，也就不用把携带的依赖关系传来传去。如果一个类持有一个对象的unsafe引用的话，必须通过某种unmanaged pointer type来实现。

对于会包含unmanaged pointer types的泛型或resilient value类型，以上规则不适用。分析会假定对泛型或resilient value type执行copy_addr，会生成一个independently-managed的值。这种扩展机制可以使得使用这种类型更方便；但并是说它就确保了语言的安全性，最终责任还是在用户身上。

0xA Copy-on-Write Representation

Copy-on-Write (COW)数据结构是通过对象引用来实现的，在数据修改时copy一份数据，并建立新的引用。

SIL中典型的COW修改操作序列如下：

```
 (%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
```

从COW数据结构load操作如下：

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

immutable属性意味着用ref_element_addr和ref_tail_addr指令loading值，如果指令的操作数相同，那么可以认为是等效的。换句话说，只要作为操作数的buffer引用相同，则可以保证buffer的属性在两个ref_element / tail_addr [immutable]之间不会发生修改。甚至该buffer如果逃逸（escape）到其他未知函数，以上也成立。

在上面的示例中，因为两个ref_element_addr指令的操作数都是相同的%immutable_buffer，%value2和%value1相等。从概念上讲，COW buffer对象的内容可以看作是与缓buffer引用相同的静态（不可变）SSA值的一部分。

通过begin_cow_mutation和end_cow_mutation指令可以将COW值的生存期严格分为可变和不可变区域：

```
%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
...
```

begin_cow_mutation和end_cow_mutation都会consume其操作数，并把新的buffer作为一个@owned值进行返回。 begin_cow_mutation将编译为唯一性检查代码，而end_cow_mutation将编译为no-op（无操作空代码）。

尽管返回的buffer引用的物理指针值与操作数相同，但在SIL中生成新的buffer引用还是很重要的。它防止优化器将buffer访问从mutable的区域移动到immutable的区域，反之亦然。

因为buffer内容在概念上是buffer引用SSA值的一部分，所以每次更改buffer内容时都必须有一个新的buffer引用。

为了说明这一点，让我们看一个示例，其中一个COW值在一个循环中被更改。与标量SSA值一样，对COW buffer进行更改也将在循环头块中强制使用phi参数（为简单起见，未显示用于复制非唯一缓冲区的代码）：

```
Phi node
LLVM 中关于SSA的数据结构

The 'phi' instruction is used to implement the φ node in the SSA graph represe

ref:  http://llvm.org/docs/LangRef.html#phi-instruction
```

```
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:
```

两条相邻的begin_cow_mutation和end_cow_mutation指令不必在同一函数中。

0xB Instruction Set

Reference Counting

这些指令处理堆对象的引用计数。 strong引用类型的值具有被引用堆对象的ownership语义。 在SIL中，retain 和 release 操作必须被显式地写出来。 对值的retain 和 release可能会被移动位置，并且可能会检测它们是否按对匹配。 对值的使用，会维护一个owning retain技计数。

所有引用计数操作都定义为可以正确地处理null引用（strong，unowned,或者weak）。 一个non-null引用必须应用一个明确类型（或自类型）的有效对象。Address操作数必须是有效的且是non-null.

SIL要求引用计数操作必须是显式的，SIL的类型系统也完整地表达引用的效力（strength）。

```
FYI
reference strength 引用效力
笔者理解为一种语义上引用的关联程度，具体有strong，weak，unowned
```

这有以下的一些好处：

1. 类型安全：原生层是@weak和@unowned的引用，不会在SIL层输出为strong引用。
2. 一致性：当引用在内存中，像copy_addr和destroy_addr这类指令所操作的地址的具体类型，已经包含了正确的语义信息，因此就不需要一些特殊的变体或者flags来描述了。
3. 易工具化：SIL可以直接表示用户对于引用效力的意图，这使得开发一个内存使用分析器（memo profiler）变得简单。原则上，只要对SIL添加一些额外的限制和扩展，甚至可以去掉所有引用计数指令，而基于类型信息实现一个GC（垃圾内存回收器）。