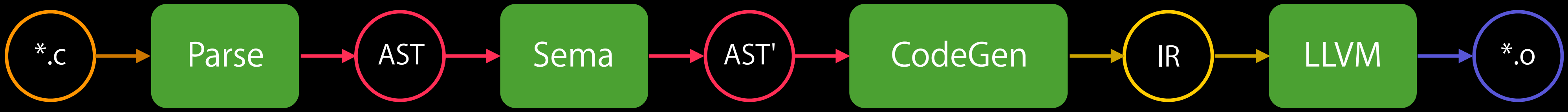
Swift Intermediate Language

A high level IR to complement LLVM

Joe Groff and Chris Lattner

Why SIL?

Clang


*.c

Parse

AST

Sema

AST'

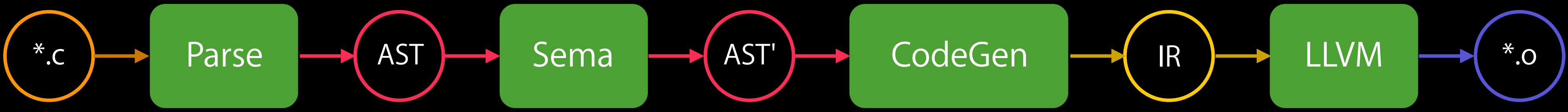
CodeGen

IR

LLVM

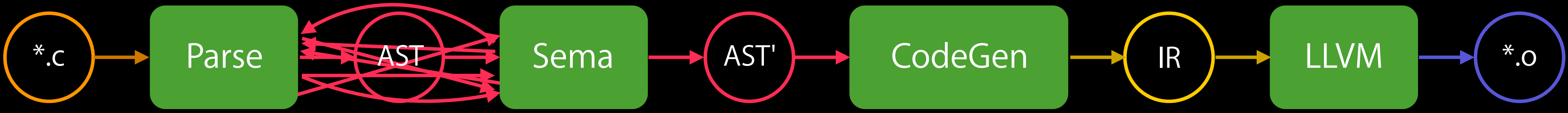
*.o

Clang

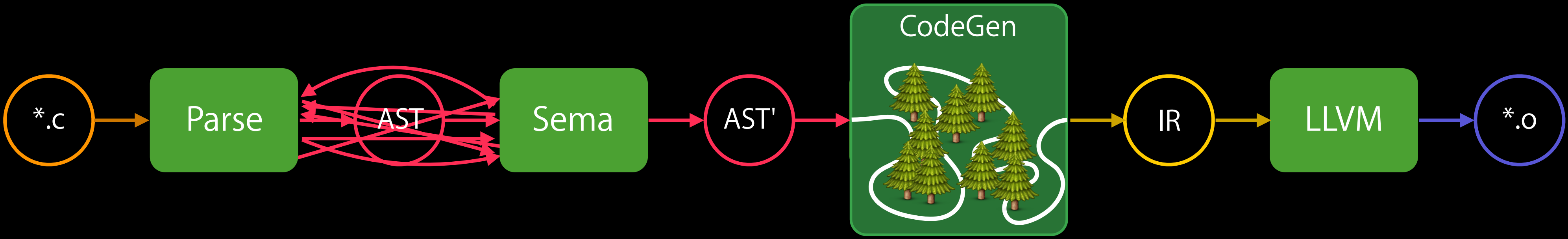


*.c → Parse → AST → Sema → AST' → CodeGen → IR → LLVM → *.o

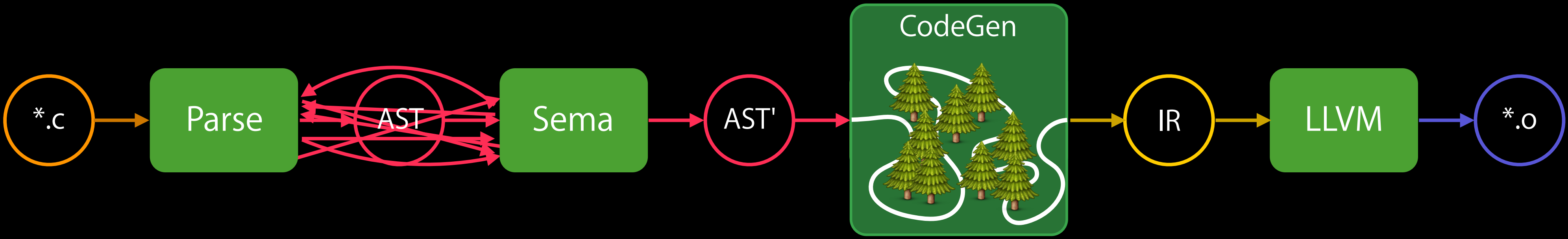
Clang



Clang



Clang



*.c

Parse

AST

Sema

AST'

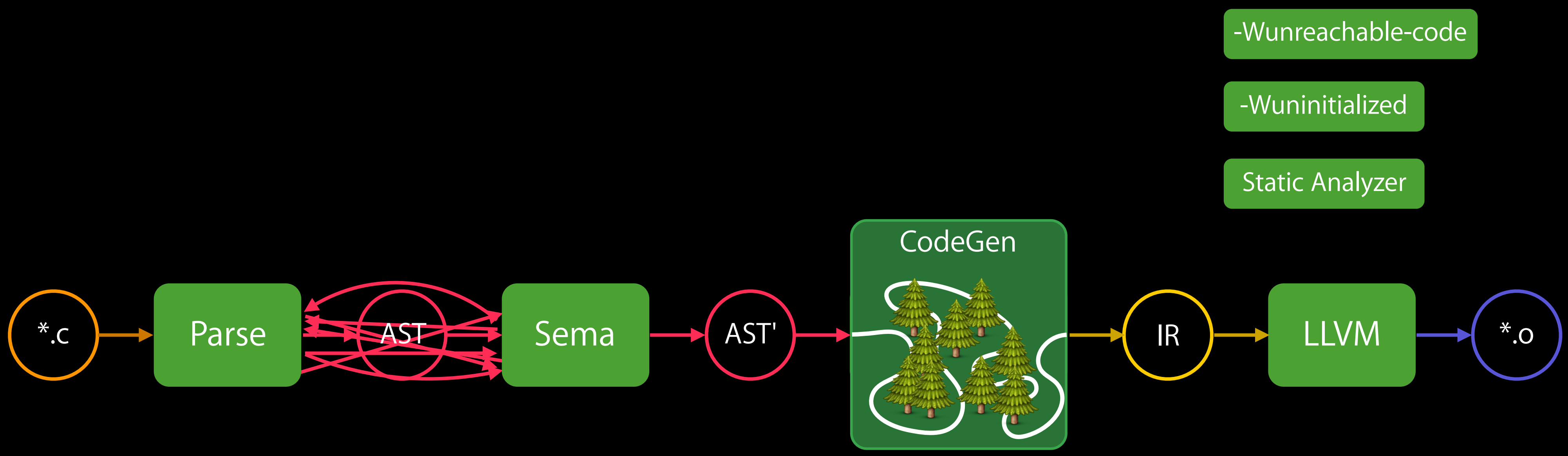
CodeGen

IR

LLVM

*.o

Clang



-Wunreachable-code

-Wuninitialized

Static Analyzer

*.c

Parse

AST

Sema

AST'

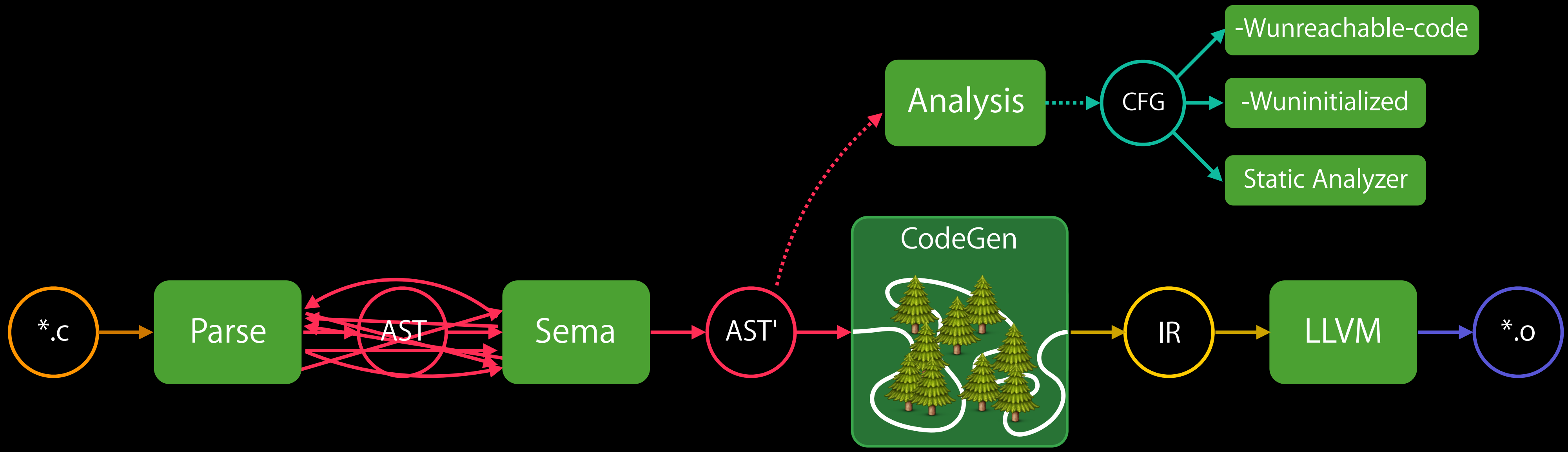
CodeGen

IR

LLVM

*.o

Clang


-Wunreachable-code
Analysis
CFG
-Wuninitialized
Static Analyzer
CodeGen
*.c
Parse
AST
Sema
AST'
IR
LLVM
*.o

Clang

Wide abstraction gap between source and LLVM IR

IR isn't suitable for source-level analysis

CFG lacks fidelity

CFG is off the hot path

Duplicated effort in CFG and IR lowering

Swift

Swift

Higher-level language

Swift

Higher-level language

- Move more of the language into code

Swift

Higher-level language

- Move more of the language into code
- Protocol-based generics

Swift

Higher-level language

- Move more of the language into code
- Protocol-based generics

Safe language

Swift

Higher-level language

- Move more of the language into code
- Protocol-based generics

Safe language

- Uninitialized vars, unreachable code should be compiler errors

Swift

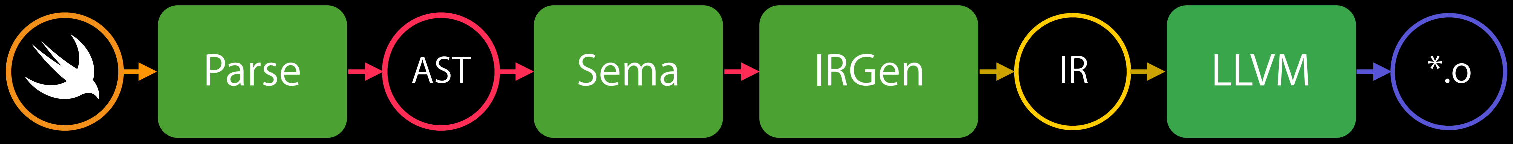
Higher-level language

- Move more of the language into code
- Protocol-based generics

Safe language

- Uninitialized vars, unreachable code should be compiler errors
- Bounds and overflow checks

Swift



Parse

AST

Sema

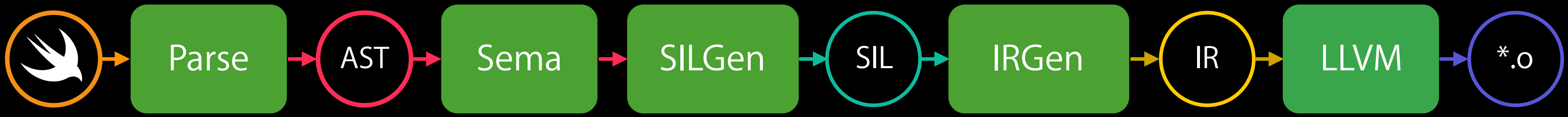
IRGen

IR

LLVM

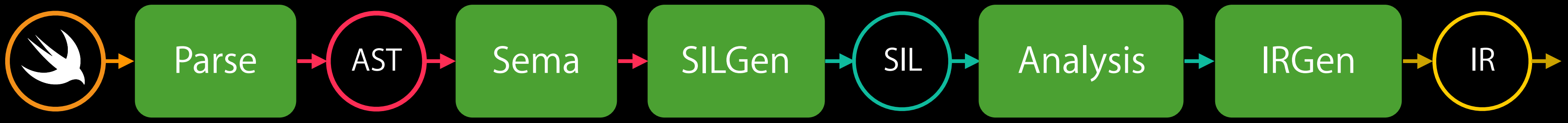
*.o

Swift



Parse → AST → Sema → SILGen → SIL → IRGen → IR → LLVM → *.o

Swift



Parse → AST → Sema → SILGen → SIL → Analysis → IRGen → IR →

SIL

Fully represents program semantics

Designed for both code generation and analysis

Sits on the hot path of the compiler pipeline

Bridges the abstraction gap between source and LLVM

Design of SIL

Fibonacci

```swift
func fibonacci(lim: Int) {
  var a = 0, b = 1
  while b < lim {
    print(b)
    (a, b) = (b, a + b)
  }
}
```

Fibonacci

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
```

Fibonacci

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
```

Fibonacci

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
```

High-Level Type System

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
```

High-Level Type System

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
```

High-Level Type System

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
```

High-Level Type System

High-Level Type System

Keeps machine-level type layout abstract

High-Level Type System

Keeps machine-level type layout abstract

Type-related information is implicit

High-Level Type System

Keeps machine-level type layout abstract

Type-related information is implicit

- TBAA

High-Level Type System

Keeps machine-level type layout abstract

Type-related information is implicit

- TBAA
- Type parameters for generic specialization

High-Level Type System

Keeps machine-level type layout abstract

Type-related information is implicit

- TBAA
- Type parameters for generic specialization
- Classes and protocol conformances for devirtualization

High-Level Type System

Keeps machine-level type layout abstract

Type-related information is implicit

- TBAA
- Type parameters for generic specialization
- Classes and protocol conformances for devirtualization

Strongly-typed IR helps validate compiler correctness

Builtins

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
```

Builtins

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
```

Builtins

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1


  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Builtins

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1


  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Builtins

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1


  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Builtins

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1


  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Builtins

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1


  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Builtins

Builtins

Builtins opaquely represent types and operations of the layer below SIL

Builtins

Builtins opaquely represent types and operations of the layer below SIL

Swift's standard library implements user-level interfaces on top of builtins

Builtins

Builtins opaquely represent types and operations of the layer below SIL

Swift's standard library implements user-level interfaces on top of builtins

```
struct Int { var value: Builtin.Int64 }
struct Bool { var value: Builtin.Int1 }
func ==(lhs: Int, rhs: Int) -> Bool {
  return Bool(value: Builtin.icmp_eq_Word(lhs.value, rhs.value))
}
```

Builtins

Builtins opaquely represent types and operations of the layer below SIL

Swift's standard library implements user-level interfaces on top of builtins

```
struct Int { var value: Builtin.Int64 }
struct Bool { var value: Builtin.Int1 }
func ==(lhs: Int, rhs: Int) -> Bool {
  return Bool(value: Builtin.icmp_eq_Word(lhs.value, rhs.value))
}
```

SIL is intentionally ignorant of:

- Machine-level type layout
- Arithmetic, comparison, etc. machine-level operations

Literal Instructions

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Word, 0
  %b0 = integer_literal $Builtin.Word, 1


  %lt = builtin "icmp_lt_Word"(%b: $Builtin.Word, %lim: $Builtin.Word): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Literal Instructions

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Word, 0
  %b0 = integer_literal $Builtin.Word, 1


  %lt = builtin "icmp_lt_Word"(%b: $Builtin.Word, %lim: $Builtin.Word): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Literal Instructions

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Word, 0
  %b0 = integer_literal $Builtin.Word, 1


  %lt = builtin "icmp_lt_Word"(%b: $Builtin.Word, %lim: $Builtin.Word): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Literal Instructions

Literal Instructions

More uniform IR representation

Literal Instructions

More uniform IR representation

- No `Value`/`Constant` divide

Literal Instructions

More uniform IR representation

- No `Value`/`Constant` divide

All instructions carry source location information for diagnostics

Literal Instructions

More uniform IR representation

- No `Value`/`Constant` divide

All instructions carry source location information for diagnostics

Especially important for numbers, which need to be statically checked for overflow

Phi Nodes?

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1


  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
```

Phi Nodes?

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
  br loop(%a0: $Builtin.Int64, %b0: $Builtin.Int64)
loop(%a: $Builtin.Int64, %b: $Builtin.Int64):
  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
body:
  %b1 = struct $Swift.Int (%b: $Builtin.Int64)
  apply %print(%b1) : $(Swift.Int) -> ()
  %c = builtin "add_Int64"(%a: $Builtin.Int64, %b: $Builtin.Int64): $Builtin.Int64
  br loop(%b: $Builtin.Int64, %c: $Builtin.Int64)
exit:
  %unit = tuple ()
  return %unit: $()
}
```

Phi Nodes?

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
  br loop(%a0: $Builtin.Int64, %b0: $Builtin.Int64)
loop(%a: $Builtin.Int64, %b: $Builtin.Int64):
  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
body:
  %b1 = struct $Swift.Int (%b: $Builtin.Int64)
  apply %print(%b1) : $(Swift.Int) -> ()
  %c = builtin "add_Int64"(%a: $Builtin.Int64, %b: $Builtin.Int64): $Builtin.Int64
  br loop(%b: $Builtin.Int64, %c: $Builtin.Int64)
exit:
  %unit = tuple ()
  return %unit: $()
}
```

Basic Block Arguments

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
  br loop(%a0: $Builtin.Int64, %b0: $Builtin.Int64)
loop(%a: $Builtin.Int64, %b: $Builtin.Int64):
  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
body:
  %b1 = struct $Swift.Int (%b: $Builtin.Int64)
  apply %print(%b1) : $(Swift.Int) -> ()
  %c = builtin "add_Int64"(%a: $Builtin.Int64, %b: $Builtin.Int64): $Builtin.Int64
  br loop(%b: $Builtin.Int64, %c: $Builtin.Int64)
exit:
  %unit = tuple ()
  return %unit: $()
}
```

Basic Block Arguments

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
  br loop(%a0: $Builtin.Int64, %b0: $Builtin.Int64)
loop(%a: $Builtin.Int64, %b: $Builtin.Int64):
  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
body:
  %b1 = struct $Swift.Int (%b: $Builtin.Int64)
  apply %print(%b1) : $(Swift.Int) -> ()
  %c = builtin "add_Int64"(%a: $Builtin.Int64, %b: $Builtin.Int64): $Builtin.Int64
  br loop(%b: $Builtin.Int64, %c: $Builtin.Int64)
exit:
  %unit = tuple ()
  return %unit: $()
}
```

Basic Block Arguments

Basic Block Arguments

More uniform IR representation

Basic Block Arguments

More uniform IR representation

- Entry block is no longer a special case

Basic Block Arguments

More uniform IR representation

- Entry block is no longer a special case
- No special case code for managing phis

Basic Block Arguments

More uniform IR representation

- Entry block is no longer a special case
- No special case code for managing phis

Provides natural notation for conditional defs

Basic Block Arguments

More uniform IR representation

- Entry block is no longer a special case
- No special case code for managing phis

Provides natural notation for conditional defs

```
entry:
  %s = invoke @mayThrowException(), label %success, label %failure
success:
  /* can only use %s here */
failure:
  %e = landingpad
```

Basic Block Arguments

More uniform IR representation

- Entry block is no longer a special case
- No special case code for managing phis

Provides natural notation for conditional defs

```
entry:
  invoke @mayThrowException(), label %success, label %failure
success(%s):
  /* can only use %s here */
failure(%e):
```

Fibonacci

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
  br loop(%a0: $Builtin.Int64, %b0: $Builtin.Int64)
loop(%a: $Builtin.Int64, %b: $Builtin.Int64):
  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
  cond_br %lt: $Builtin.Int1, body, exit
body:
  %b1 = struct $Swift.Int (%b: $Builtin.Int64)
  apply %print(%b1) : $(Swift.Int) -> ()
  %c = builtin "add_Int64"(%a: $Builtin.Int64, %b: $Builtin.Int64): $Builtin.Int64
  br loop(%b: $Builtin.Int64, %c: $Builtin.Int64)
exit:
  %unit = tuple ()
  return %unit: $()
}
```

Method Lookup

```
entry(%c: $SomeClass):
  %foo = class_method %c: $SomeClass, #SomeClass.foo : $(SomeClass) -> ()
  apply %foo(%c) : $(SomeClass) -> ()
```

Method Lookup

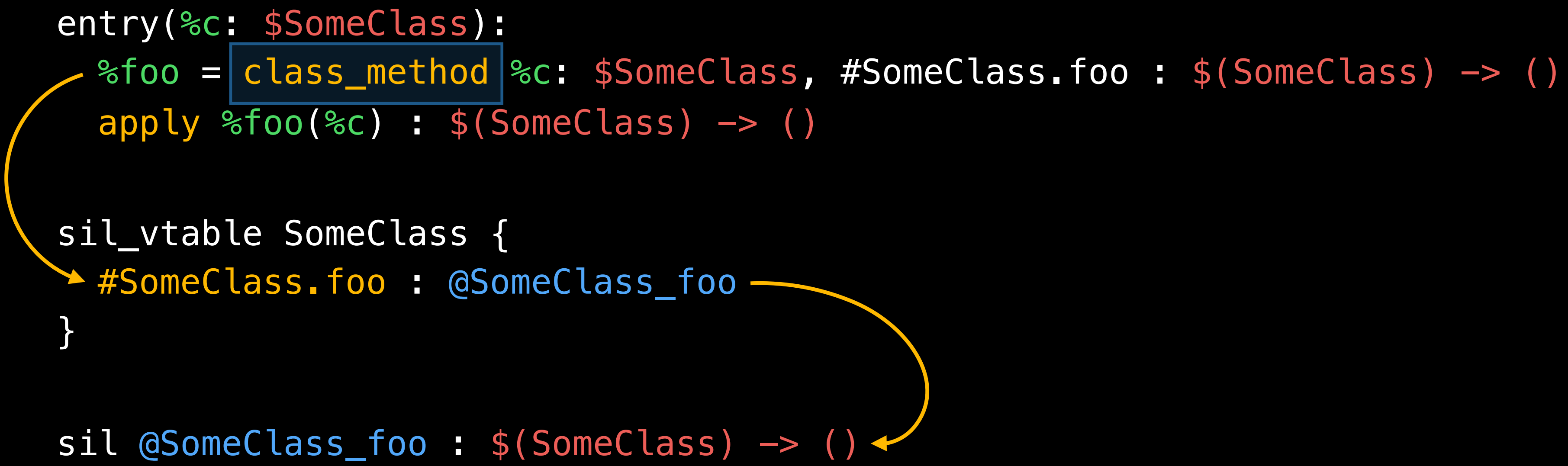
```
entry(%c: $SomeClass):
  %foo = class_method %c: $SomeClass, #SomeClass.foo : $(SomeClass) -> ()
  apply %foo(%c) : $(SomeClass) -> ()
```

Method Lookup

```
entry(%c: $SomeClass):
  %foo = class_method %c: $SomeClass, #SomeClass.foo : $(SomeClass) -> ()
  apply %foo(%c) : $(SomeClass) -> ()

sil_vtable SomeClass {
  #SomeClass.foo : @SomeClass_foo
}
```

Method Lookup



```
entry(%c: $SomeClass):
  %foo = class_method %c: $SomeClass, #SomeClass.foo : $(SomeClass) -> ()
  apply %foo(%c) : $(SomeClass) -> ()

sil_vtable SomeClass {
  #SomeClass.foo : @SomeClass_foo
}

sil @SomeClass_foo : $(SomeClass) -> ()
```

Method Lookup

```
entry(%c: $SomeClass):
  %foo = function_ref @SomeClass_foo : $(SomeClass) -> ()
  apply %foo(%c) : $(SomeClass) -> ()
```

Method Lookup

```
entry(%x: $T, %y: $T):
  %plus = witness_method $T, #Addable.+ : $<U: Addable> (U, U) -> U
  %z = apply %plus<T>(%x, %y) : $<U: Addable> (U, U) -> U
```

Method Lookup

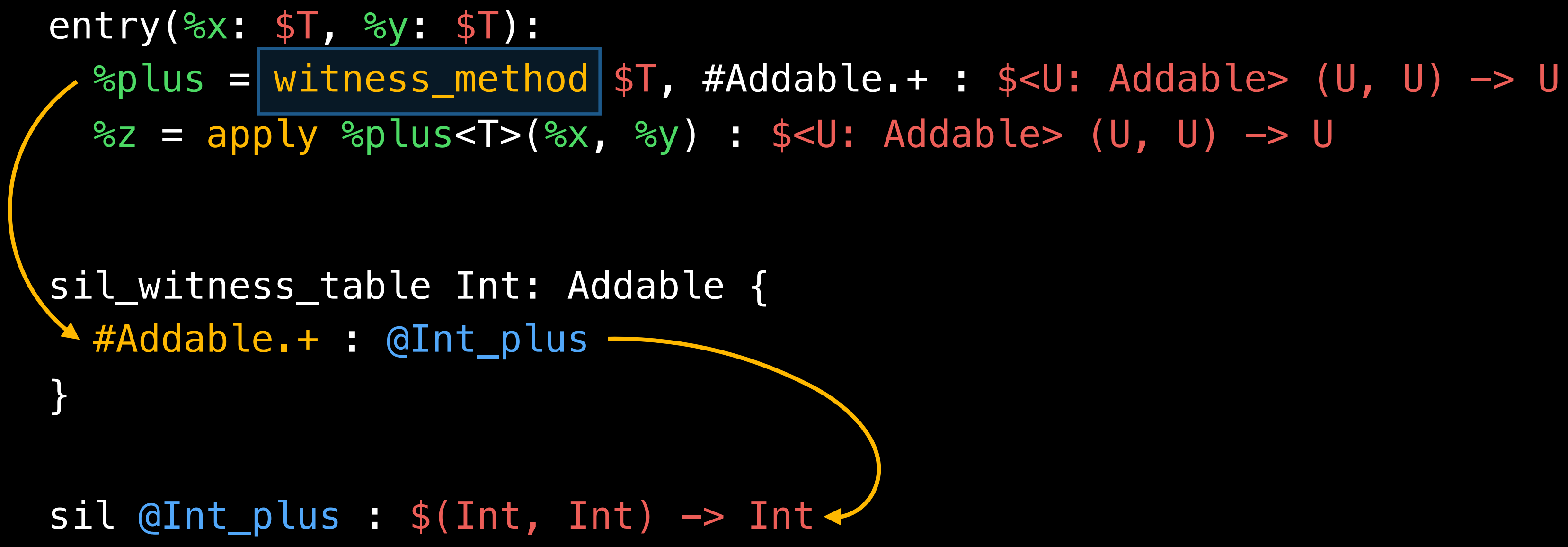
```
entry(%x: $T, %y: $T):
  %plus = witness_method $T, #Addable.+ : $<U: Addable> (U, U) -> U
  %z = apply %plus<T>(%x, %y) : $<U: Addable> (U, U) -> U
```

Method Lookup

```
entry(%x: $T, %y: $T):
  %plus = witness_method $T, #Addable.+ : $<U: Addable> (U, U) -> U
  %z = apply %plus<T>(%x, %y) : $<U: Addable> (U, U) -> U


sil_witness_table Int: Addable {
  #Addable.+ : @Int_plus
}
```

Method Lookup



```
entry(%x: $T, %y: $T):
  %plus = witness_method $T, #Addable.+ : $<U: Addable> (U, U) -> U
  %z = apply %plus<T>(%x, %y) : $<U: Addable> (U, U) -> U


sil_witness_table Int: Addable {
  #Addable.+ : @Int_plus
}

sil @Int_plus : $(Int, Int) -> Int
```

Method Lookup

```
entry(%x: $Int, %y: $Int):
  %plus = function_ref @Int_plus : $(Int, Int) -> Int
  %z = apply %plus(%x, %y) : $(Int, Int) -> Int
```

Memory Allocation

Memory Allocation

```
%stack = alloc_stack $Int
```

Memory Allocation

```
%stack = alloc_stack $Int
store %x to %stack: $*Int
%y = load %stack: $*Int
```

Memory Allocation

```
%stack = alloc_stack $Int
store %x to %stack: $*Int
%y = load %stack: $*Int
dealloc_stack %stack: $*Int
```

Memory Allocation

```
%stack = alloc_stack $Int
store %x to %stack: $*Int
%y = load %stack: $*Int
dealloc_stack %stack: $*Int

%box = alloc_box $Int
```

Memory Allocation

```
%stack = alloc_stack $Int
store %x to %stack: $*Int
%y = load %stack: $*Int
dealloc_stack %stack: $*Int

%box = alloc_box $Int

%object = alloc_ref $SomeClass
```

Memory Allocation

```
%stack = alloc_stack $Int
store %x to %stack: $*Int
%y = load %stack: $*Int
dealloc_stack %stack: $*Int

%box = alloc_box $Int

%object = alloc_ref $SomeClass

strong_retain %object : $SomeClass
strong_release %object : $SomeClass
```

Control Flow

Control Flow

```
br loop
cond_br %flag: $Builtin.Int1, yes, no
return %x: $Int
unreachable
```

Control Flow

```
  br loop
  cond_br %flag: $Builtin.Int1, yes, no
  return %x: $Int
  unreachable

  switch_enum %e: $Optional<Int>, case #Optional.Some: some,
                                  case #Optional.None: none
some(%x: $Int):
```

Control Flow

```
  br loop
  cond_br %flag: $Builtin.Int1, yes, no
  return %x: $Int
  unreachable

  switch_enum %e: $Optional<Int>, case #Optional.Some: some,
                                  case #Optional.None: none
some(%x: $Int):

  checked_cast_br %c: $BaseClass, $DerivedClass, success, failure
success(%d: $DerivedClass):
```

Program Failure

Program Failure

```
%result = builtin "sadd_with_overflow_Int64"
   (%x : $Builtin.Int64, %y : $Builtin.Int64) : $(Builtin.Int64, Builtin.Int1)
%overflow = tuple_extract %result, 1
```

Program Failure

```
  %result = builtin "sadd_with_overflow_Int64"
     (%x : $Builtin.Int64, %y : $Builtin.Int64) : $(Builtin.Int64, Builtin.Int1)
  %overflow = tuple_extract %result, 1
  cond_br %overflow : $Builtin.Int1, fail, cont
cont:
  %z = tuple_extract %result, 0

  /* ... */

fail:
  builtin "int_trap"()
  unreachable
```

Program Failure

```
%result = builtin "sadd_with_overflow_Int64"
   (%x : $Builtin.Int64, %y : $Builtin.Int64) : $(Builtin.Int64, Builtin.Int1)
%overflow = tuple_extract %result, 1


%z = tuple_extract %result, 0
```

Program Failure

```
%result = builtin "sadd_with_overflow_Int64"
   (%x : $Builtin.Int64, %y : $Builtin.Int64) : $(Builtin.Int64, Builtin.Int1)
%overflow = tuple_extract %result, 1

cond_fail %overflow : $Builtin.Int1
%z = tuple_extract %result, 0
```

Swift’s use of SIL

Two Phases of SIL Passes

Early SIL:

- Data flow sensitive lowering
- SSA-based diagnostics
- “Guaranteed” optimizations

Late SIL:

- Performance optimizations
- Serialization
- LLVM IRGen

Early SIL

Many individual passes:

Early SIL

Many individual passes:

- Mandatory inlining
- Capture promotion
- Box-to-stack promotion
- inout argument deshadowing
- Diagnose unreachable code
- Definitive initialization
- Guaranteed memory optimizations
- Constant folding / overflow diagnostics

Early SIL

Many individual passes:

- Mandatory inlining
- Capture promotion
- Box-to-stack promotion
- inout argument deshadowing
- Diagnose unreachable code
- Definitive initialization
- Guaranteed memory optimizations
- Constant folding / overflow diagnostics

Problems we'll look at:

- Diagnosing Overflow
- Enabling natural closure semantics with memory safety
- Removing requirement for default construction

Diagnosing Overflow

```
let v = Int8(127)+1
```

Arithmetic overflow is guaranteed to trap in Swift

- Not undefined behavior
- Not 2’s complement (unless explicitly using &+ operator)

Diagnosing Overflow

```
let v = Int8(127)+1
```

Arithmetic overflow is guaranteed to trap in Swift

- Not undefined behavior
- Not 2’s complement (unless explicitly using &+ operator)

How can we statically diagnose overflow?

- … and produce a useful error message?

Output of SILGen

```
let v = Int8(127)+1
```

```
 %1 = integer_literal $Builtin.Int2048, 127
 %2 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
 %4 = apply [transparent] %2(%1) : $(Builtin.Int2048) -> Int8

 %5 = integer_literal $Builtin.Int2048, 1
 %6 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
 %8 = apply [transparent] %6(%5) : $(Builtin.Int2048) -> Int8

%9 = function_ref @“Swift.+” : $(Int8, Int8) -> Int8
%10 = apply [transparent] %0(%4, %8) : $(Int8, Int8) -> Int8
debug_value %10 : $Int8  // let v
```

Output of SILGen

```
let v = Int8(127)+1
```

```
%1 = integer_literal $Builtin.Int2048, 127
%2 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
%4 = apply [transparent] %2(%1) : $(Builtin.Int2048) -> Int8

%5 = integer_literal $Builtin.Int2048, 1
%6 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
%8 = apply [transparent] %6(%5) : $(Builtin.Int2048) -> Int8

%9 = function_ref @“Swift.+” : $(Int8, Int8) -> Int8
%10 = apply [transparent] %0(%4, %8) : $(Int8, Int8) -> Int8
debug_value %10 : $Int8  // let v
```

Output of SILGen

```
let v = Int8(127)+1
```

```
%1 = integer_literal $Builtin.Int2048, 127
%2 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
%4 = apply [transparent] %2(%1) : $(Builtin.Int2048) -> Int8

%5 = integer_literal $Builtin.Int2048, 1
%6 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
%8 = apply [transparent] %6(%5) : $(Builtin.Int2048) -> Int8

%9 = function_ref @“Swift.+” : $(Int8, Int8) -> Int8
%10 = apply [transparent] %0(%4, %8) : $(Int8, Int8) -> Int8
debug_value %10 : $Int8  // let v
```

Output of SILGen

```
let v = Int8(127)+1
```

```
%1 = integer_literal $Builtin.Int2048, 127
%2 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
%4 = apply [transparent] %2(%1) : $(Builtin.Int2048) -> Int8

%5 = integer_literal $Builtin.Int2048, 1
%6 = function_ref @“Swift.Int8.init" : $(Builtin.Int2048) -> Int8
%8 = apply [transparent] %6(%5) : $(Builtin.Int2048) -> Int8

%9 = function_ref @“Swift.+” : $(Int8, Int8) -> Int8
%10 = apply [transparent] %0(%4, %8) : $(Int8, Int8) -> Int8
debug_value %10 : $Int8  // let v
```

After mandatory inlining

```
let v = Int8(127)+1
```

After mandatory inlining

```
let v = Int8(127)+1
```

```
%0 = integer_literal $Builtin.Int8, 127
%4 = integer_literal $Builtin.Int8, 1
%11 = builtin "sadd_with_overflow_Int8"(%0 : $Builtin.Int8, %4 : $Builtin.Int8)

%12 = tuple_extract %11 : $(Builtin.Int8, Builtin.Int1), 1
cond_fail %12 : $Builtin.Int1

%13 = tuple_extract %11 : $(Builtin.Int8, Builtin.Int1), 0
%15 = struct $Int8 (%13 : $Builtin.Int8)
debug_value %15 : $Int8  // let v
```

After mandatory inlining

```
let v = Int8(127)+1
```

```
%0 = integer_literal $Builtin.Int8, 127
%4 = integer_literal $Builtin.Int8, 1
%11 = builtin "sadd_with_overflow_Int8"(%0 : $Builtin.Int8, %4 : $Builtin.Int8)

%12 = tuple_extract %11 : $(Builtin.Int8, Builtin.Int1), 1
cond_fail %12 : $Builtin.Int1

%13 = tuple_extract %11 : $(Builtin.Int8, Builtin.Int1), 0
%15 = struct $Int8 (%13 : $Builtin.Int8)
debug_value %15 : $Int8  // let v
```

After mandatory inlining

```
let v = Int8(127)+1
```

```
%0 = integer_literal $Builtin.Int8, 127
%4 = integer_literal $Builtin.Int8, 1
%11 = builtin "sadd_with_overflow_Int8"(%0 : $Builtin.Int8, %4 : $Builtin.Int8)

%12 = tuple_extract %11 : $(Builtin.Int8, Builtin.Int1), 1
cond_fail %12 : $Builtin.Int1

%13 = tuple_extract %11 : $(Builtin.Int8, Builtin.Int1), 0
%15 = struct $Int8 (%13 : $Builtin.Int8)
debug_value %15 : $Int8  // let v
```

Diagnostic constant folding

```
let v = Int8(127)+1
```

```
%0 = integer_literal $Builtin.Int8, -128
%1 = integer_literal $Builtin.Int1, -1
%2 = tuple (%0 : $Builtin.Int8, %1 : $Builtin.Int1)    // folded “sadd_overflow”

cond_fail %1 : $Builtin.Int1                           // unconditional failure
```

Diagnostic constant folding

```
let v = Int8(127)+1
```

```
%0 = integer_literal $Builtin.Int8, -128
%1 = integer_literal $Builtin.Int1, -1
%2 = tuple (%0 : $Builtin.Int8, %1 : $Builtin.Int1)    // folded “sadd_overflow”

cond_fail %1 : $Builtin.Int1                           // unconditional failure
```

Diagnostic constant folding

```
let v = Int8(127)+1
```

```
%0 = integer_literal $Builtin.Int8, -128
%1 = integer_literal $Builtin.Int1, -1
%2 = tuple (%0 : $Builtin.Int8, %1 : $Builtin.Int1)    // folded “sadd_overflow”

cond_fail %1 : $Builtin.Int1                           // unconditional failure
```

Each SIL instruction maintains full location information:

- Pointer back to AST node it came from
- Including SIL inlining information

```
t.swift:2:20: error: arithmetic operation '127 + 1' (on type 'Int8') results in an overflow
  let v = Int8(127) + 1
          ~~~~~~~~~ ^ ~
```

Local variable optimization

Memory safety with closures provides challenges:

- Closures can capture references to local variables
- Closure lifetime is not limited to a stack discipline

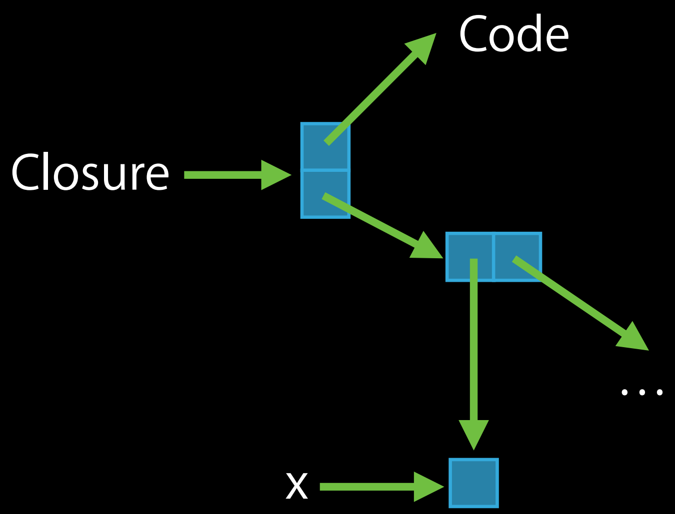
```
func doSomething() -> Int {
  var x = 1
  takeClosure { x = 2 }
  return x
}
```

Local variable optimization

Memory safety with closures provides challenges:

- Closures can capture references to local variables
- Closure lifetime is not limited to a stack discipline

```
func doSomething() -> Int {
  var x = 1
  takeClosure { x = 2 }
  return x
}
```



Solution:

Semantic model is for all stack variables to be on the heap

Local variables after SILGen

SILGen emits all local ‘var’iables as heap boxes with alloc_box

```
func f() -> Int {
  var x = 42

  return x
}
```

Local variables after SILGen

SILGen emits all local ‘var’iables as heap boxes with alloc_box

```
func f() -> Int {
  var x = 42

  return x
}
```

```
%0 = alloc_box $Int  // var x
%4 = ...
store %4 to %0#1 : $*Int
```

Local variables after SILGen

SILGen emits all local ‘var’iables as heap boxes with alloc_box

```
func f() -> Int {
  var x = 42

  return x
}
```

```
%0 = alloc_box $Int  // var x
%4 = ...
store %4 to %0#1 : $*Int

%6 = load %0#1 : $*Int
strong_release %0#0
return %6 : $Int
```

Local variables after SILGen

SILGen emits all local ‘var’iables as heap boxes with alloc_box

```swift
func f() -> Int {
  var x = 42

  return x
}
```

```
%0 = alloc_box $Int  // var x
%4 = ...
store %4 to %0#1 : $*Int

%6 = load %0#1 : $*Int
strong_release %0#0
return %6 : $Int
```

Box-to-stack promotes heap boxes to stack allocations

All closure captures are by reference

- Not acceptable to leave them on the heap!

Promotion eliminates byref capture

Safe to promote to by-value capture in many cases:

- … e.g. when no mutations happen after closure formation

This enables the captured value to be promoted to the stack/registers

Promotion eliminates byref capture

Safe to promote to by-value capture in many cases:

… e.g. when no mutations happen after closure formation

This enables the captured value to be promoted to the stack/registers

```
var x = …
x += 42

arr1 = arr2.map { elt in elt+x }
```

Promotion eliminates byref capture

Safe to promote to by-value capture in many cases:

… e.g. when no mutations happen after closure formation

This enables the captured value to be promoted to the stack/registers

```
var x = …
x += 42

arr1 = arr2.map { elt in elt+x }
```

➡

```
var x = …
x += 42
let x2 = x
arr1 = arr2.map { elt in elt+x2 }
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x




%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%9 = function_ref @“closure1” : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int
%10 = partial_apply %9(%2#0, %2#1) : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%9 = function_ref @“closure1” : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int
%10 = partial_apply %9(%2#0, %2#1) : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%9 = function_ref @“closure1” : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int
%10 = partial_apply %9(%2#0, %2#1) : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array



sil @“closure1” {
bb0(%0 : $Int, %1 : $Builtin.NativeObject, %2 : $*Int):
  debug_value %0 : $Int  // let elt
  %4 = load %2 : $*Int
  ...
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%9 = function_ref @“closure1” : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int
%10 = partial_apply %9(%2#0, %2#1) : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array



sil @“closure1” {
bb0(%0 : $Int, %1 : $Builtin.NativeObject, %2 : $*Int):
  debug_value %0 : $Int  // let elt
  %4 = load %2 : $*Int
  ...
```

Naive SIL for by-ref closure capture

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%9 = function_ref @“closure1” : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int
%10 = partial_apply %9(%2#0, %2#1) : $(Int, @owned Builtin.NativeObject, @inout Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array
```

```
sil @“closure1” {
bb0(%0 : $Int, %1 : $Builtin.NativeObject, %2 : $*Int):
  debug_value %0 : $Int  // let elt
  %4 = load %2 : $*Int
  ...
```

SIL after capture promotion

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x




%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array
```

SIL after capture promotion

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%4 = load %2#1 : $*Int
%7 = function_ref @“closure1” : $(Int, Int) -> Int
%10 = partial_apply %7(%4) : $(Int, Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array
```

SIL after capture promotion

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%4 = load %2#1 : $*Int
%7 = function_ref @“closure1” : $(Int, Int) -> Int
%10 = partial_apply %7(%4) : $(Int, Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array
```

SIL after capture promotion

```
arr = arr.map { elt in elt+x }
```

```
%2 = alloc_box $Int  // var x

%4 = load %2#1 : $*Int
%7 = function_ref @“closure1” : $(Int, Int) -> Int
%10 = partial_apply %7(%4) : $(Int, Int) -> Int

%11 = function_ref @“Array.map” : $((Int) -> Int, Array) -> Array
%12 = apply %11(%10, %0) : $((Int) -> Int, Array) -> Array


sil @“closure1” {
bb0(%0 : $Int, %1 : $Int):
  debug_value %0 : $Int  // let elt
  debug_value %1 : $Int  // var x
  ...
```

Definitive Initialization

Problem:

- Not all values can be default initialized

```
func testDI(cond : Bool) {
  var v : SomeClass

  if cond {
    v = SomeClass(1234)
  } else {
    v = SomeClass(4321)
  }

  v.foo()
}
```

Definitive Initialization

Problem:

- Not all values can be default initialized

Desires:

- Don’t want magic numbers for primitive types
- Want to allow flexible initialization patterns

```
func testDI(cond : Bool) {
  var v : SomeClass

  if cond {
    v = SomeClass(1234)
  } else {
    v = SomeClass(4321)
  }

  v.foo()
}
```

Definitive Initialization

Problem:

- Not all values can be default initialized

Desires:

- Don’t want magic numbers for primitive types
- Want to allow flexible initialization patterns

Solution:

- Dataflow driven liveness analysis

```
func testDI(cond : Bool) {
  var v : SomeClass

  if cond {
    v = SomeClass(1234)
  } else {
    v = SomeClass(4321)
  }

  v.foo()
}
```

Definitive Initialization

Problem:

- Not all values can be default initialized

Desires:

- Don’t want magic numbers for primitive types
- Want to allow flexible initialization patterns

Solution:

- Dataflow driven liveness analysis

```
func testDI(cond : Bool) {
  var v : SomeClass

  if cond {
    v = SomeClass(1234)
  }


  v.foo()
}
```

error: 'v' used before being initialized

Definitive Initialization Algorithm

Check each use of value to determine:

- Guaranteed initialized
- Guaranteed uninitialized
- Initialized only on some paths

```
struct Pair {
  var a, b : Int
  init() {
    a = 42
  }
}
```

Definitive Initialization Algorithm

Check each use of value to determine:

- Guaranteed initialized
- Guaranteed uninitialized
- Initialized only on some paths

Diagnostics must be great

```
struct Pair {
  var a, b : Int
  init() {
    a = 42
  }
}
```

note: 'self.b' not initialized

error: return from initializer without initializing all stored properties

DI covers many similar cases

```swift
func test() -> Float {
  var local : (Int, Float)
  local.0 = 42
  return local.1
}
```

```swift
class Base {
  init(x : Int) {}
}

class Derived : Base {
  var x, y : Int

  init() {
    x = 42; y = 1
  }
}
```

DI covers many similar cases

```
func test() -> Float {
  var local : (Int, Float)
  local.0 = 42
  return local.1
}
```

error: 'local.1' used before being initialized

```
class Base {
  init(x : Int) {}
}

class Derived : Base {
  var x, y : Int

  init() {
    x = 42; y = 1
  }
}
```

error: super.init isn't called before returning from initializer

DI Lowering: Initialization vs Assignment

```
x = y
```

Semantics depend on data flow properties

First assignment is initialization:

- Raw memory → Valid value

Subsequent assignments are replacements:

- Valid value → Valid value

DI Lowering: Initialization vs Assignment

`x = y`

Semantics depend on data flow properties

First assignment is initialization:

- Raw memory → Valid value

```
strong_retain %y : $C
store %y to %x : $*C
```

Subsequent assignments are replacements:

- Valid value → Valid value

DI Lowering: Initialization vs Assignment

```
x = y
```

Semantics depend on data flow properties

First assignment is initialization:

- Raw memory → Valid value

```
strong_retain %y : $C
store %y to %x : $*C
```

Subsequent assignments are replacements:

- Valid value → Valid value

```
strong_retain %y : $C
%tmp = load %x : $*C
strong_release %tmp : $C
store %y to %x : $*C
```

Conditional Liveness

- Inherently a dataflow problem
  - Requires dynamic logic in some cases

- Conditional destruction too

```
func testDI(cond : Bool) {
  var c : SomeClass






  c = SomeClass(4321)    // init or assign?

  c.foo()
}
```

Conditional Liveness

- Inherently a dataflow problem
  - Requires dynamic logic in some cases
- Conditional destruction too

```
func testDI(cond : Bool) {
  var c : SomeClass

  if cond {
    c = SomeClass(1234)
  }

  c = SomeClass(4321)    // init or assign?

  c.foo()
}
```

Language-Specific IR: Retrospective

Diagnostics

Clear improvement over Clang CFG for data flow diagnostics:

- Diagnostics always up to date as language evolves
- Great location information, source level type information
- DCE before diagnostics eliminates “false” positives

IMHO Clang should pull clang::CFG (or something better) into its IRGen path

Lowering

Nice separation between SILGen and IRGen:

- SILGen handles operational lowering
- IRGen handles type lowering & concretization of the target

Lowering

Nice separation between SILGen and IRGen:

- SILGen handles operational lowering
- IRGen handles type lowering & concretization of the target

Dataflow Lowering:

- Great way to handle things like swift assignment vs initialization
- Can be emulated by generating LLVM intrinsics and lowering on IR

Performance Optimizations

Necessary for generics specialization:

- Requires full source level type system
- Specialization produces extreme changes to generated IR

Performance Optimizations

Necessary for generics specialization:

- Requires full source level type system
- Specialization produces extreme changes to generated IR

Less clear for other optimizations

- ARC Optimization, devirt, etc could all be done on IR (with tradeoffs)

Performance Optimizations

Necessary for generics specialization:

- Requires full source level type system
- Specialization produces extreme changes to generated IR

Less clear for other optimizations

- ARC Optimization, devirt, etc could all be done on IR (with tradeoffs)

Required a ton of infrastructure:

- SILCombine
- Passmanager for analyses
- …

Summary

SIL was a lot of work, but necessary given the scope of Swift
  May make sense (or not) based on your language

We're pretty happy with it…
  …but there is still a ton of work left to do

Know LLVM and use it for what it is good for
  … don't reinvent everything just for fun :-)