$$T(n) \sim O(f(n))$$

$$T(n) \leq c \cdot f(n)$$

$$T(n) \leq \cancel{c} \cdot f(n)$$

$T(n) \leq$  $T(n)$

BULLSHIT

NON-PESSIMIZATIONS IN THE SWIFT COMPILER

Árpád Goretity

@H2CO3_iOS

Demo

What?

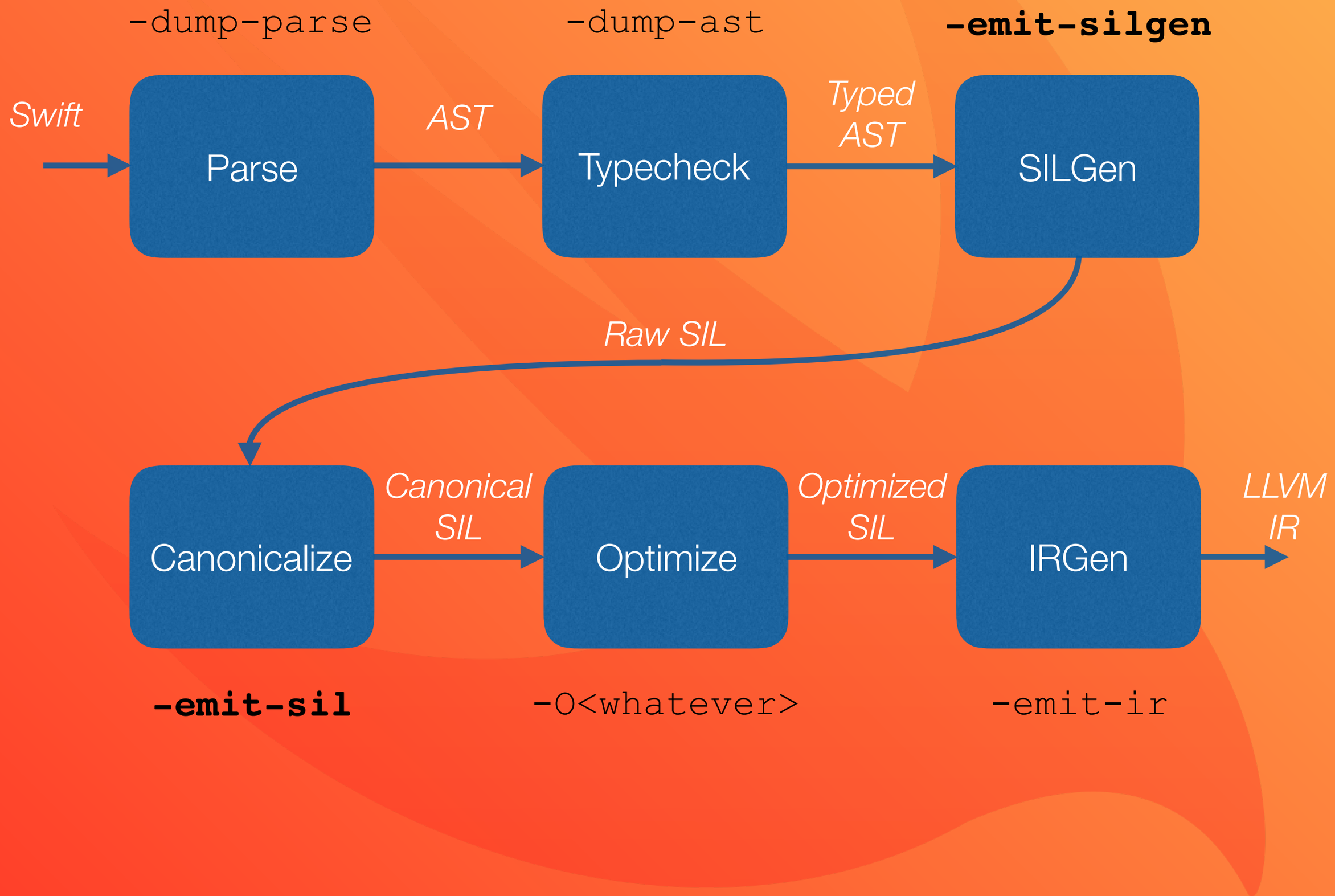
Non-pessimization

Non-pessimization

=

Canonicalization

-Onone


-dump-parse
-dump-ast
-emit-silgen
Swift
Parse
AST
Typecheck
Typed AST
SILGen
Raw SIL
Canonicalize
Canonical SIL
Optimize
Optimized SIL
IRGen
LLVM IR
-emit-sil
-O<whatever>
-emit-ir


SILGen,
Programmer
Naïve / Raw
SIL
Canonicalize

How?


Act 1
Memory

```
1func FnWithInout(inout x: Int) {
2    x = 1337
3}
```

```
sil hidden Act1.FnWithInout (inout Int) -> () {
bb0(%0 : $*Int):
  // make shadow variable
  %1 = alloc_box $Int
  // shadow = x
  copy_addr %0 to [initialization] %1#1 : $*Int

  // shadow = Int(1337)
  %5 = integer_literal $Builtin.Int2048, 1337
  %6 = apply Swift.Int.Init(%5, ...)
  assign %6 to %1#1 : $*Int

  // x = shadow
  copy_addr %1#1 to %0 : $*Int

  // return ()
  strong_release %1#0 : $@box Int
  %10 = tuple ()
  return %10 : $()
}
```

```
sil hidden Act1.FnWithInout (inout Swift.Int) -> () {
bb0(%0 : $*Int):
  %1 = integer_literal $Builtin.Int64, 1337
  %2 = struct $Int (%1 : $Builtin.Int64)
  store %2 to %0 : $*Int
  %4 = tuple ()
  return %4 : $()
}
```

```
5func CallInout() {
6    var x = 42
7    FnWithInout(&x)
8}
```

```
sil hidden Act1.CallInout () -> () {
bb0:
  // var x...
  %0 = alloc_box $Int
  // ... = 42
  %3 = integer_literal $Builtin.Int2048, 42
  %4 = apply Swift.Int.Init(%3, ...)
  store %4 to %0#1 : $*Int

  // FnWithInout(&x)
  %7 = apply Act1.FnWithInout(%0#1) : (@inout Int) -> ()

  // return ()
  strong_release %0#0 : $@box Int
  %9 = tuple ()
  return %9 : $()
}
```

```
sil hidden Act1.CallInout () -> () {
bb0:
  // var x = 42
  %0 = alloc_stack $Int
  %1 = integer_literal $Builtin.Int64, 42
  %2 = struct $Int (%1 : $Builtin.Int64)
  store %2 to %0#1 : $*Int

  // FnWithInout(&x)
  %5 = apply Act1.FnWithInout(%0#1) : (inout Swift.Int) -> ()

  // deallocate x, return unit
  %6 = tuple ()
  dealloc_stack %0#0 : $*@local_storage Int
  return %6 : $()
}
```

Act 2
ARChitecture

```
1class Foo {
2    // ...
3}
4
5func use_foo(foo: Foo) {
6}
7
8func juggle_foo() {
9    var f1 = Foo() // allocate f1
10   let f2 = f1    // strong_retain f2
11   use_foo(f2)
12}                 // strong_release f2 (end-of-scope, LIFO)
13                  // strong_release f1 (end-of-scope, LIFO)
```

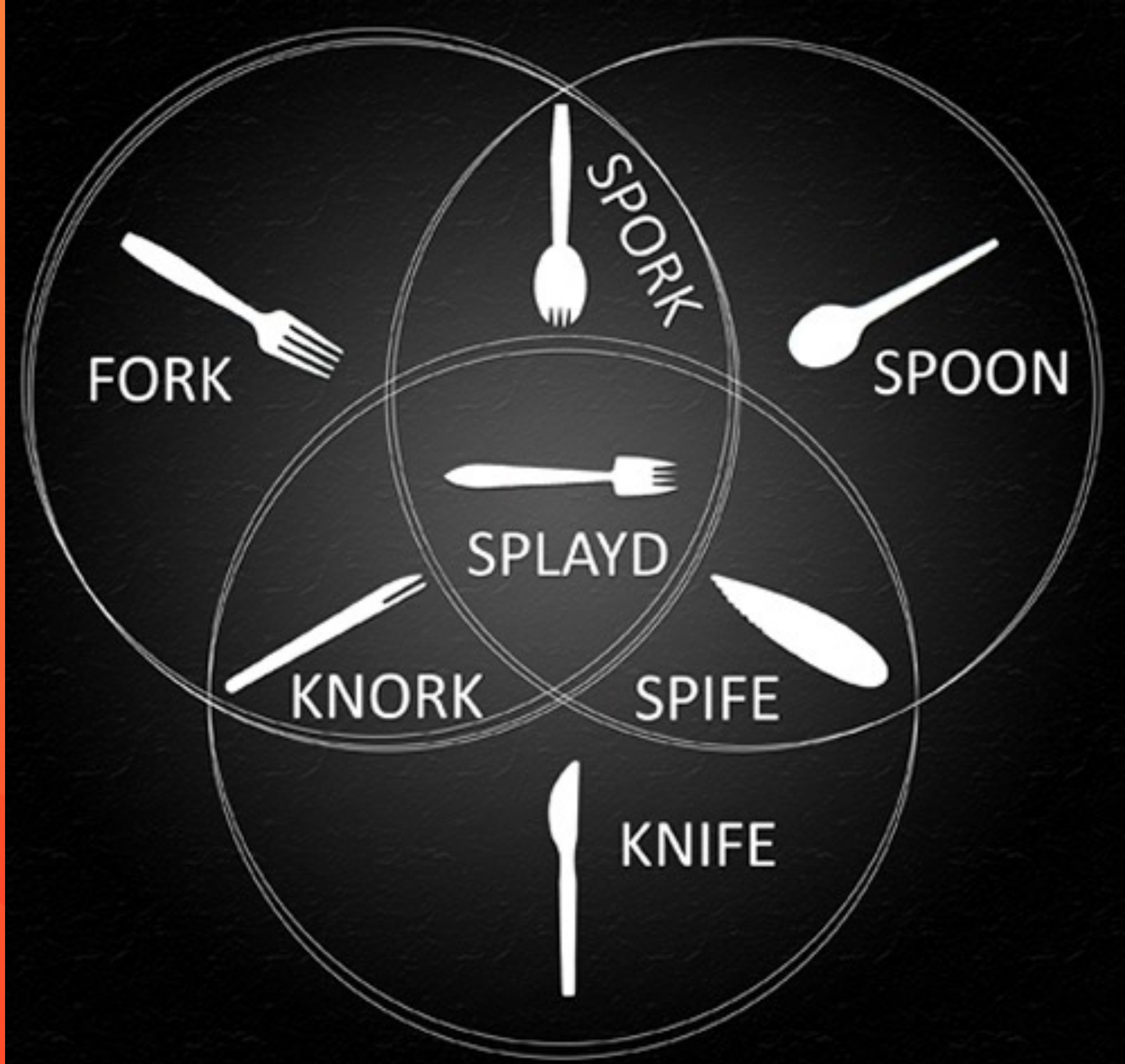
```
sil hidden arc.juggle_foo() {
bb0:
  // Line 9: Allocating: "var f1 = Foo()"
  %0 = alloc_box $Foo
  %1 = function_ref arc.Foo.__allocating_init
  %2 = metatype $@thick Foo.Type
  %3 = apply %1(%2)
  store %3 to %0#1 : $*Foo

  // Line 10: "let f2 = f1"
  %5 = load %0#1 : $*Foo
  strong_retain %5 : $Foo // assignment retains, as expected
  debug_value %5 : $Foo

  // Line 11: "use_foo(f2)"
  %8 = function_ref arc.use_foo
  strong_retain %5 : $Foo // needed because use_foo consumes refcount
  %10 = apply %8(%5)

  // Line 12: Scope exit, releasing in LIFO order
  strong_release %5 : $Foo // release f2
  strong_release %0#0 : $@box Foo // release f1

  // return unit
  %13 = tuple ()
  return %13 : $()
}
```

```
1class Foo {
2    // ...
3}
4
5func use_foo(foo: Foo) {
6}
7
8func juggle_foo() {
9    var f1 = Foo() // allocate f1
10   let f2 = f1    // strong_retain f2
11
12}                 // strong_release f2 (end-of-scope, LIFO)
13                  // strong_release f1 (end-of-scope, LIFO)
```

```
sil hidden arc.juggle_foo() {
bb0:
  // Line 9: Allocating: "var f1 = Foo()"
  %0 = alloc_box $Foo
  %1 = function_ref arc.Foo.__allocating_init
  %2 = metatype $@thick Foo.Type
  %3 = apply %1(%2)
  store %3 to %0#1 : $*Foo

  // Line 10: "let f2 = f1"
  %5 = load %0#1 : $*Foo
  // Here would be a "strong_retain %5"!
  debug_value %5 : $Foo

  // Line 12: Scope exit -> Here would be a "strong_release %5"!
  // But it's immediately preceded by a strong_retain,
  // so they were both zapped. But %0 must still be released.
  strong_release %0#0 : $@box Foo

  // return unit
  %8 = tuple ()
  return %8 : $()
}
```

Act 3

Unified Functions



Visual Basic

MATLAB

COBOL

JS

php

DELPHI

```
1func incrementResult(fn: () -> Int) -> Int {
2    return fn() + 1
3}
4
5func getAnswer() -> Int {
6    return 42
 7}
 8
 9func doStuff() {
10    getAnswer()
11    incrementResult(getAnswer)
12}
```

```
// $@convention (@owned @callee_owned () -> Int) -> Int
sil hidden Act3.incrementResult (() -> Swift.Int) -> Swift.Int // ...

sil hidden Act3.getAnswer : $@convention(thin) () -> Int // ...

sil hidden Act3.doStuff () -> () {
bb0:
  // Direct Call
  %0 = function_ref Act3.getAnswer
  %1 = apply %0() : $@convention(thin) () -> Int

  // Wrapped Call
  %2 = function_ref Act3.incrementResult
  %3 = function_ref Act3.getAnswer
  %4 = thin_to_thick_function %3
       : $@convention(thin) () -> Int to $@callee_owned () -> Int
  %5 = apply %2(%4)
       : $@convention(thin) (@owned @callee_owned () -> Int) -> Int

  %6 = tuple ()
  return %6 : $()
}
```

So What?

1. Constants Matter

1. Constants Matter
2. Canonicalization Improves Architecture

1. Constants Matter
2. Canonicalization Improves Architecture
3. Optimize Fundamental Abstractions

H2CO3/NPSwift

Questions?

Árpád Goretity

@H2CO3_iOS