Debug Info for Concurrency

Adrian Prantl

LLVM Developers' Meeting | Apple Inc. | 2023

The State Of Debugging in 2022

Robert O'Callahan, Keynote, SPLASH'22

> Problem #4d: Language Features
>
> Consider async/await in C++ and Rust. Functions containing "yield" are compiled to interruptible state machines with local variables packed into structs.
>
> Preserving a full-fidelity debugging experience would require significant debugger and compiler support. This hasn't been done.

https://youtu.be/yCK0-vWmAsk?t=1766

Swift Concurrency

https://www.swift.org/blog/swift-5.5-released/

Swift Concurrency

Async/await, Structured Concurrency, Actors

https://www.swift.org/blog/swift-5.5-released/

Swift Concurrency

Async/await, Structured Concurrency, Actors

Introduced in 2021

https://www.swift.org/blog/swift-5.5-released/

Swift Concurrency

Async/await, Structured Concurrency, Actors

Introduced in 2021

Full Debugger Support in LLDB

- Feels like debugging synchronous code
- Backtraces, Stepping
- Variable Inspection

https://www.swift.org/blog/swift-5.5-released/

Agenda

How async/await code breaks every assumption debuggers make

How to produce async backtraces

How to generate async debug info

Synchronous Code

```
func fox🦊(parameter : String) {
    let local_var = parameter
    print("calling toad")
    toad🐸()
    print("calling hare")
    hare🐰()
    print(local_var)
}

func toad🐸() { print("ribbit") }
func hare🐰() { print("rabbit") }
```

Stack

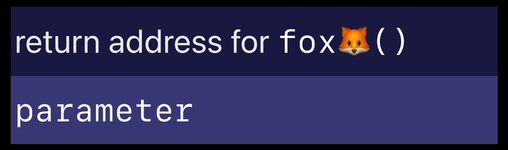
return address for `fox🦊()`

Synchronous Code

```
func fox🦊(parameter : String) {
    let local_var = parameter
    print("calling toad")
    toad🐸()
    print("calling hare")
    hare🐰()
    print(local_var)
}

func toad🐸() { print("ribbit") }
func hare🐰() { print("rabbit") }
```

Stack

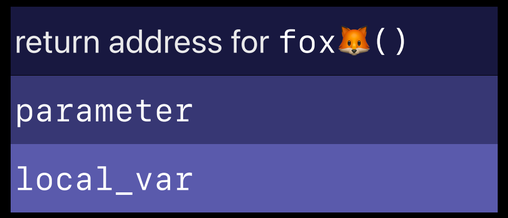


Synchronous Code

```
func fox🦊(parameter : String) {
    let local_var = parameter
    print("calling toad")
    toad🐸()
    print("calling hare")
    hare🐰()
    print(local_var)
}

func toad🐸() { print("ribbit") }
func hare🐰() { print("rabbit") }
```

Stack

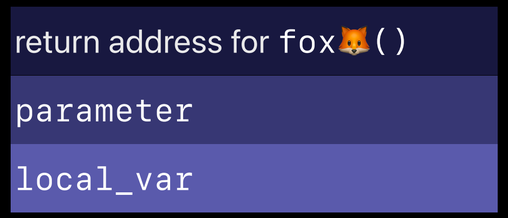


Synchronous Code

```
func fox🦊(parameter : String) {
    let local_var = parameter
    print("calling toad")
    toad🐸()
    print("calling hare")
    hare🐰()
    print(local_var)
}

func toad🐸() { print("ribbit") }
func hare🐰() { print("rabbit") }
```

Stack

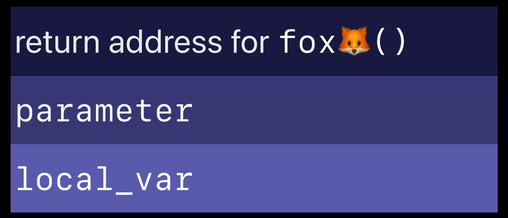


Synchronous Code

```
func fox🦊(parameter : String) {
    let local_var = parameter
    print("calling toad")
    toad🐸()
    print("calling hare")
    hare🐰()
    print(local_var)
}

func toad🐸() { print("ribbit") }
func hare🐰() { print("rabbit") }
```

Stack

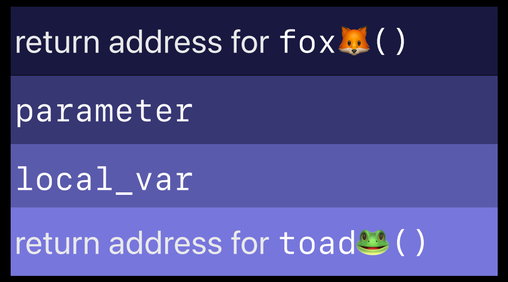


Synchronous Code

```
func fox🦊(parameter : String) {
    let local_var = parameter
    print("calling toad")
    toad🐸()
    print("calling hare")
    hare🐰()
    print(local_var)
}

func toad🐸() { print("ribbit") }
func hare🐰() { print("rabbit") }
```

Stack



Synchronous Code

```
func fox🦊(parameter : String) {
    let local_var = parameter
    print("calling toad")
    toad🐸()
    print("calling hare")
    hare🐰()
    print(local_var)
}

func toad🐸() { print("ribbit") }
func hare🐰() { print("rabbit") }
```

Synchronous Code

```
func fox🦊(parameter : String)       {
    let local_var = parameter
    print("calling toad")
          toad🐸()
    print("calling hare")
          hare🐰()
    print(local_var)
}

func toad🐸()       { print("ribbit") }
func hare🐰()       { print("rabbit") }
```

Asynchronous Code

Fundamentally changes execution model and compilation pipeline

```
func fox🦊(parameter : String) async {
    let local_var = parameter
    print("calling toad")
    await toad🐸()
    print("calling hare")
    await hare🐰()
    print(local_var)
}

func toad🐸() async { print("ribbit") }
func hare🐰() async { print("rabbit") }
```

Asynchronous Code

Fundamentally changes execution model and compilation pipeline

```
func fox🦊(parameter : String) async {
    let local_var = parameter
    print("calling toad")
                toad🐸()



    print("calling hare")
                hare🐰()



    print(local_var)
}


func toad🐸(              ) async { print("ribbit") }
func hare🐰(              ) async { print("rabbit") }
```

Asynchronous Code

Fundamentally changes execution model and compilation pipeline

```
func fox🦊#1(              ) async {
    let local_var = parameter
    print("calling toad")
                toad🐸()
}

func fox🦊#2(              ) async {
    print("calling hare")
                hare🐰()
}

func fox🦊#3(              ) async {
    print(local_var)

}

func toad🐸(              ) async { print("ribbit") }
func hare🐰(              ) async { print("rabbit") }
```

Asynchronous Code

```
func fox🦊#1(              ) async {
    let local_var = parameter
    print("calling toad")
                toad🐸()
}

func fox🦊#2(              ) async {
    print("calling hare")
                hare🐰()
}

func fox🦊#3(              ) async {
    print(local_var)

}

func toad🐸(              ) async { print("ribbit") }
func hare🐰(              ) async { print("rabbit") }
```

Functions are broken up at await boundaries (llvm::CoroSplitter)

Asynchronous Code

Debuggers hate this one trick!

```
func fox🦊#1(              ) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(              ) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(              ) async {
    print(local_var)
    task_switch                .continuation()
}

func toad🐸(             ) async { print("ribbit") }
func hare🐰(             ) async { print("rabbit") }
```

Functions are broken up at await boundaries (llvm::CoroSplitter)

Every funclet ends in a tail call or task_switch

Asynchronous Code

Debuggers hate this one trick!

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Functions are broken up at await boundaries (llvm::CoroSplitter)

Every funclet ends in a tail call or task_switch

Parameters are packed into async_context heap object

Heap Data Structure

Inside the async_context

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Heap Data Structure

Inside the async_context

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Function Argument

pointer to async_context

Heap Data Structure

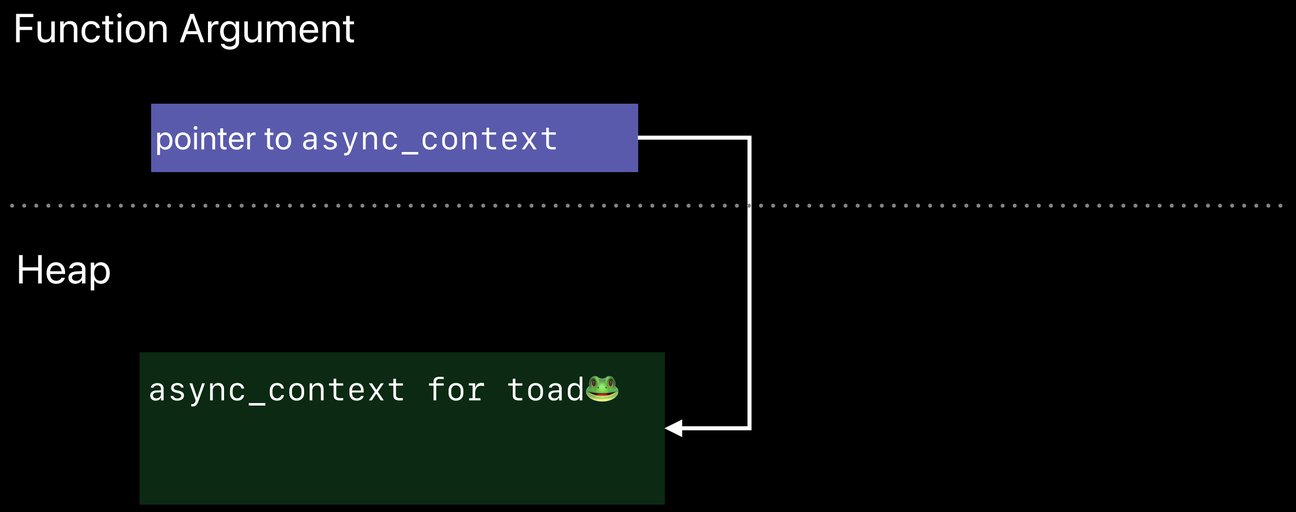
Inside the async_context

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```



Heap Data Structure

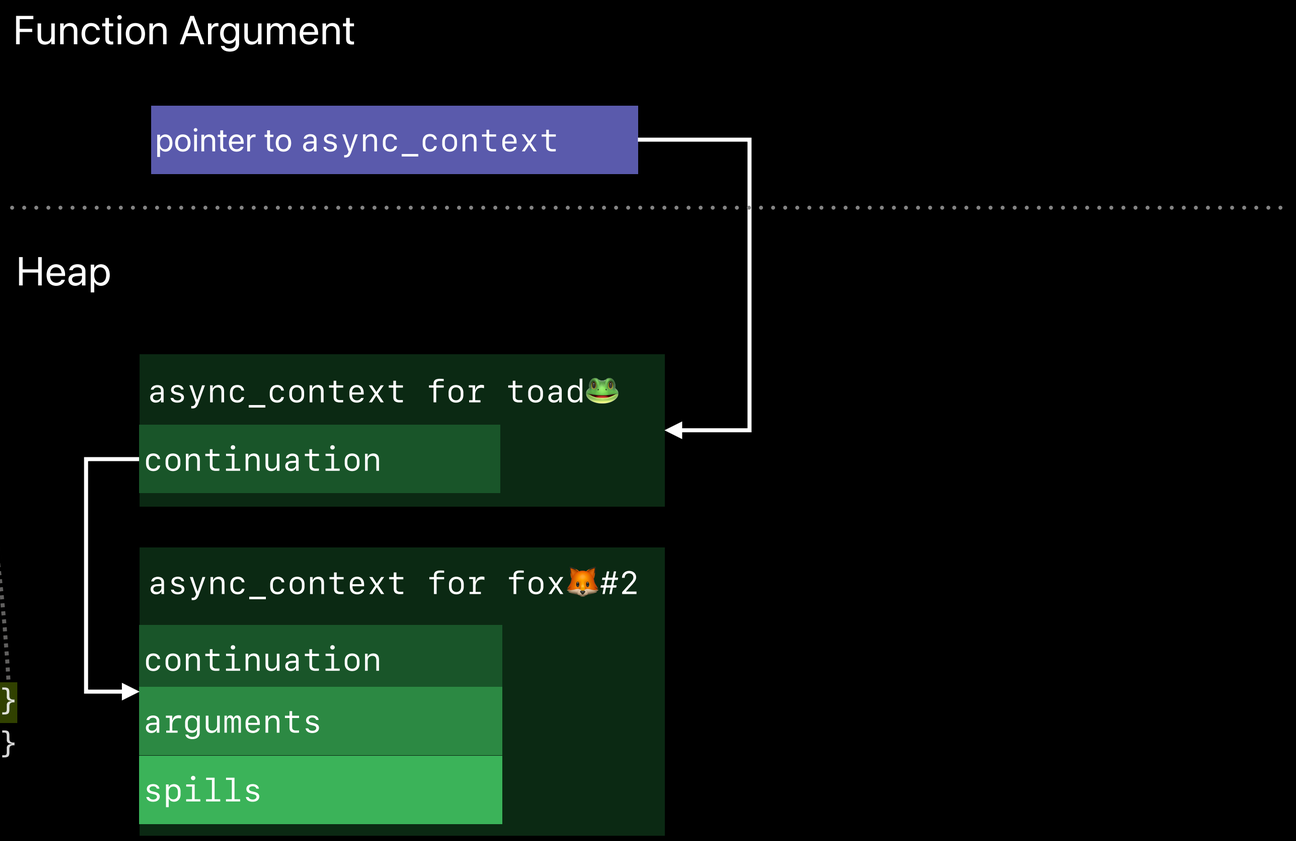
Inside the async_context

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```



Heap Data Structure

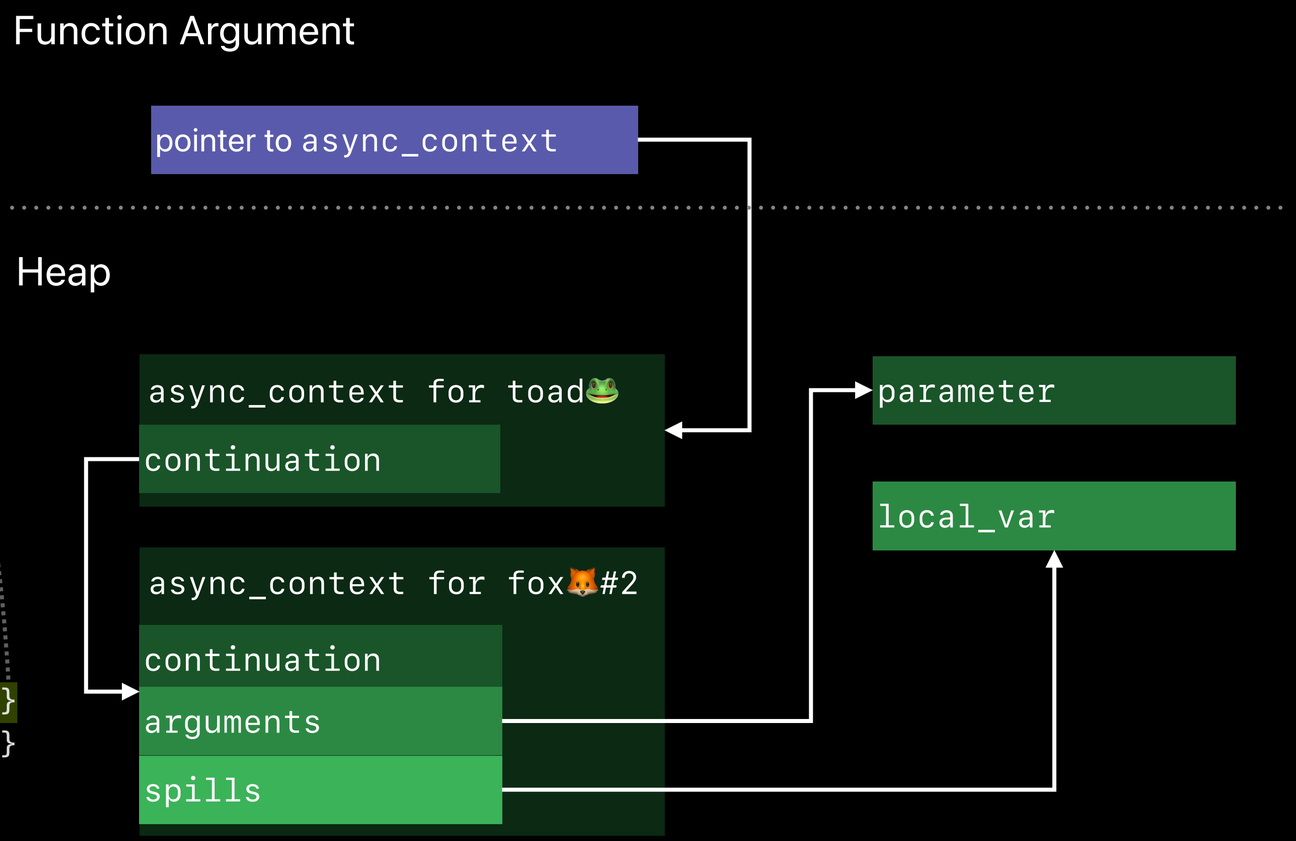
Inside the async_context

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```



Heap Data Structure

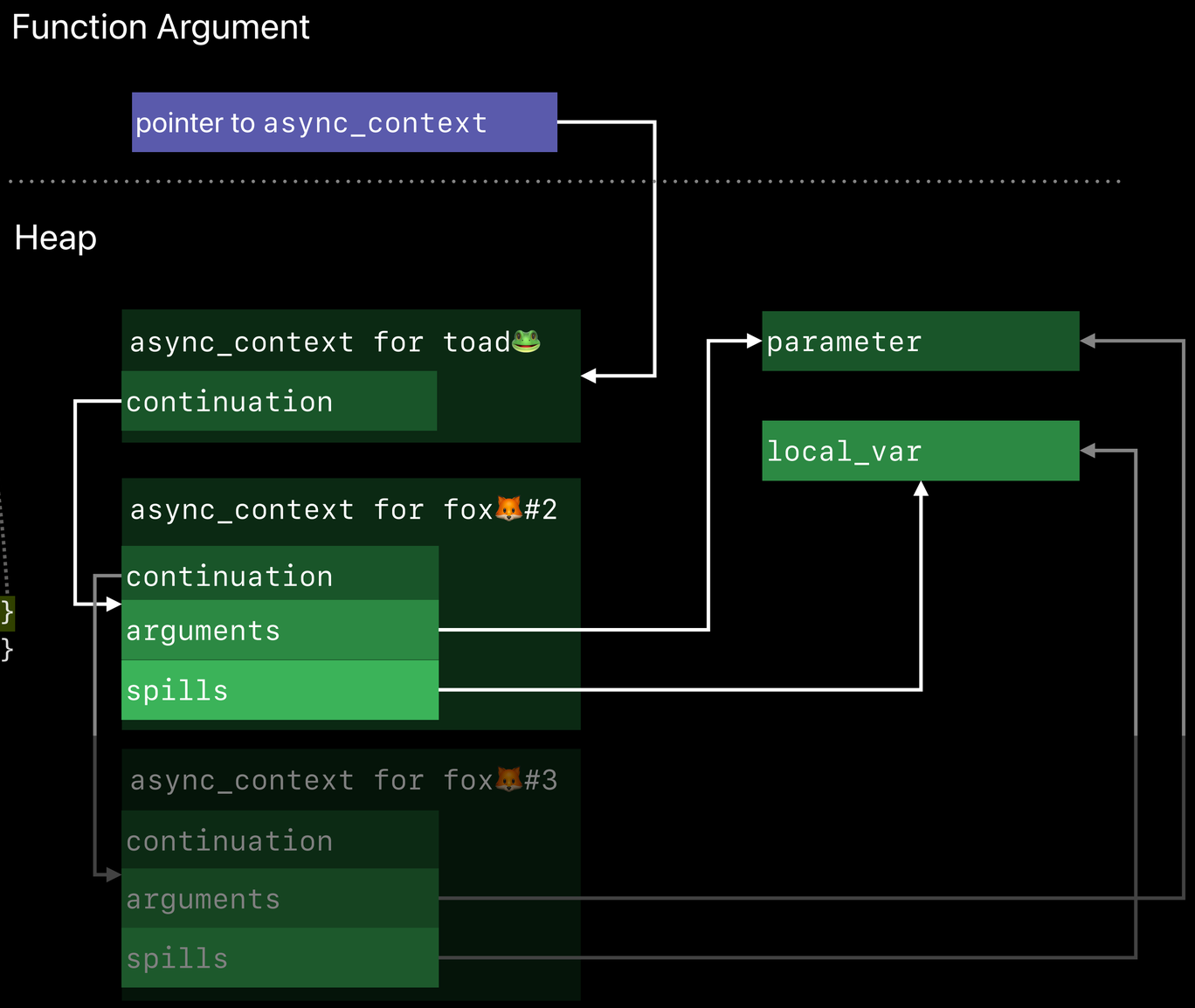
Inside the async_context

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```



Backtraces

When produced by unwinding the call stack

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Backtraces

When produced by unwinding the call stack

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

```
(lldb-without-swift-plugin) bt
* thread #2, queue = 'com.apple.root.default-qos.cooperative', stop reason = breakpoint 1.1
  * frame #0: 0x0000000100003cf0 Animals`toad🐸() at main.swift:17:6
    frame #1: 0x00000002244b8fd8 libswift_Concurrency.dylib`swift::runJobInEstablishedExecutorContext(swift::Job*) + 416
    frame #2: 0x00000002244ba19c libswift_Concurrency.dylib`swift_job_runImpl(swift::Job*, swift::ExecutorRef) + 72
    frame #3: 0x000000010053e8e4 libdispatch.dylib`_dispatch_root_queue_drain + 404
    frame #4: 0x000000010053f4f4 libdispatch.dylib`_dispatch_worker_thread2 + 188
    frame #5: 0x0000000100005fd60 libsystem_pthread.dylib`_pthread_wqthread + 228
```

Backtraces

Virtual backtraces in LLDB

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Backtraces

Virtual backtraces in LLDB

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Programmer's mental model: Backtrace is where execution came from

Backtraces

Virtual backtraces in LLDB

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Programmer's mental model: Backtrace is where execution came from

Really, it's where it's jumping (returning) to next

Backtraces

Virtual backtraces in LLDB

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Programmer's mental model: Backtrace is where execution came from

Really, it's where it's jumping (returning) to next

Async continuations also point to where execution goes next

Backtraces

Virtual backtraces in LLDB

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Programmer's mental model: Backtrace is where execution came from

Really, it's where it's jumping (returning) to next

Async continuations also point to where execution goes next

Debugger can follow continuation chain to produce a virtual backtrace

Backtraces

Virtual backtraces in LLDB

```
func fox🦊#1(async_context) async {
    let local_var = parameter
    print("calling toad")
    task_switch toad🐸()
}

func fox🦊#2(async_context) async {
    print("calling hare")
    task_switch hare🐰()
}

func fox🦊#3(async_context) async {
    print(local_var)
    task_switch async_context.continuation()
}

func toad🐸(async_context) async { print("ribbit") }
func hare🐰(async_context) async { print("rabbit") }
```

Programmer's mental model: Backtrace is where execution came from

Really, it's where it's jumping (returning) to next

Async continuations also point to where execution goes next

Debugger can follow continuation chain to produce a virtual backtrace

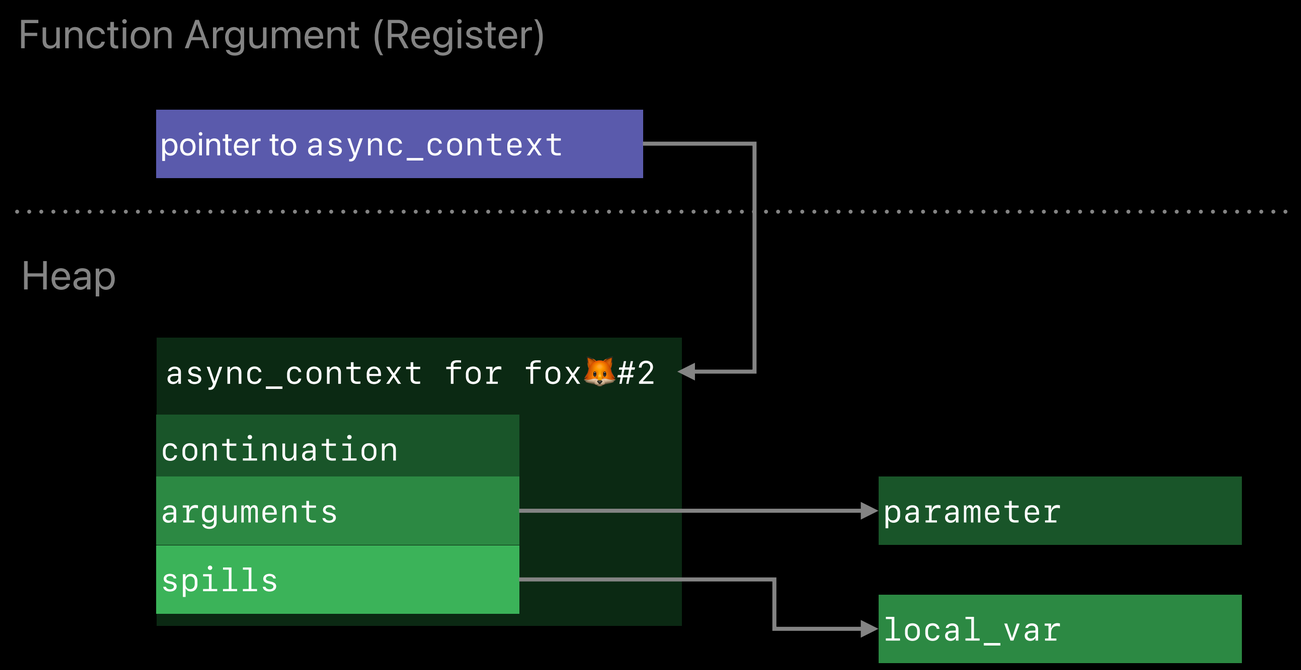
```
(lldb) bt
* thread #2, queue = 'com.apple.root.default-qos.cooperative', stop reason = breakpoint 1.2
  * frame #0: 0x0000000100003d2c Animals`toad🐸() at main.swift:17:6
    frame #1: 0x0000000100003988 Animals`fox🦊(parameter="") at main.swift:13
```

Variables

Variables

Heap Data Structure async_context

- Function Parameters
- Spilled Variables



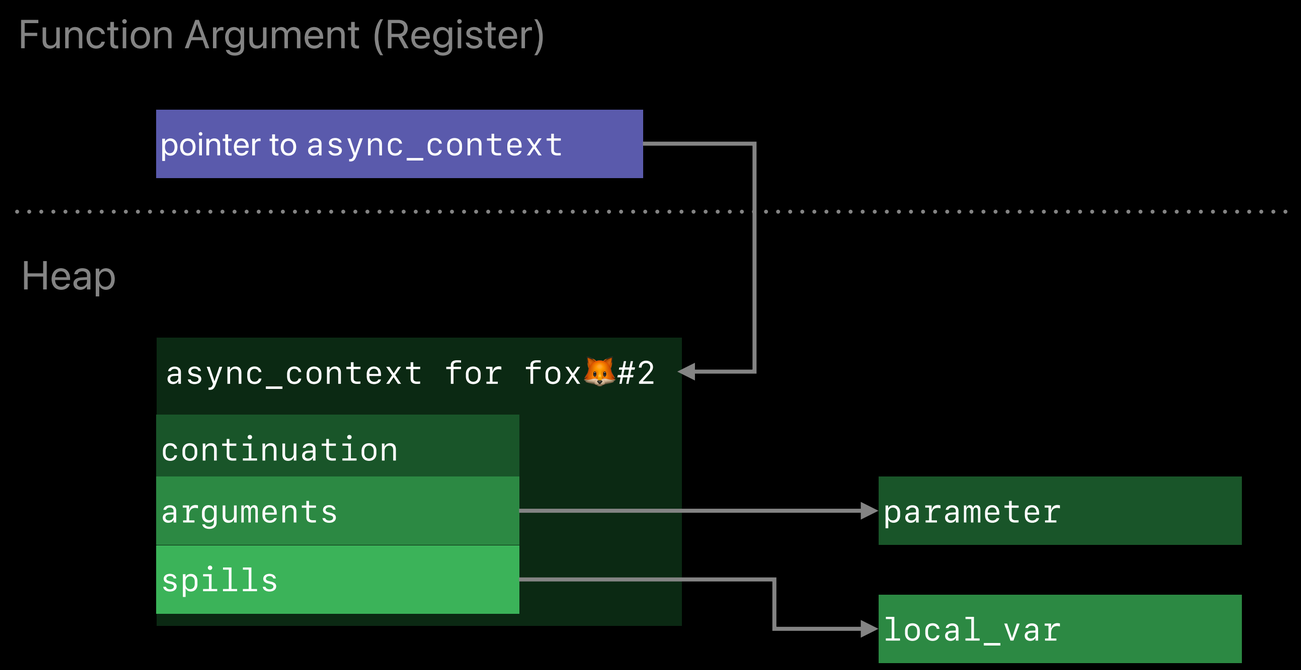
Variables

Heap Data Structure async_context

- Function Parameters
- Spilled Variables

Dedicated Register for Address of async_context

Guaranteed by Swift ABI



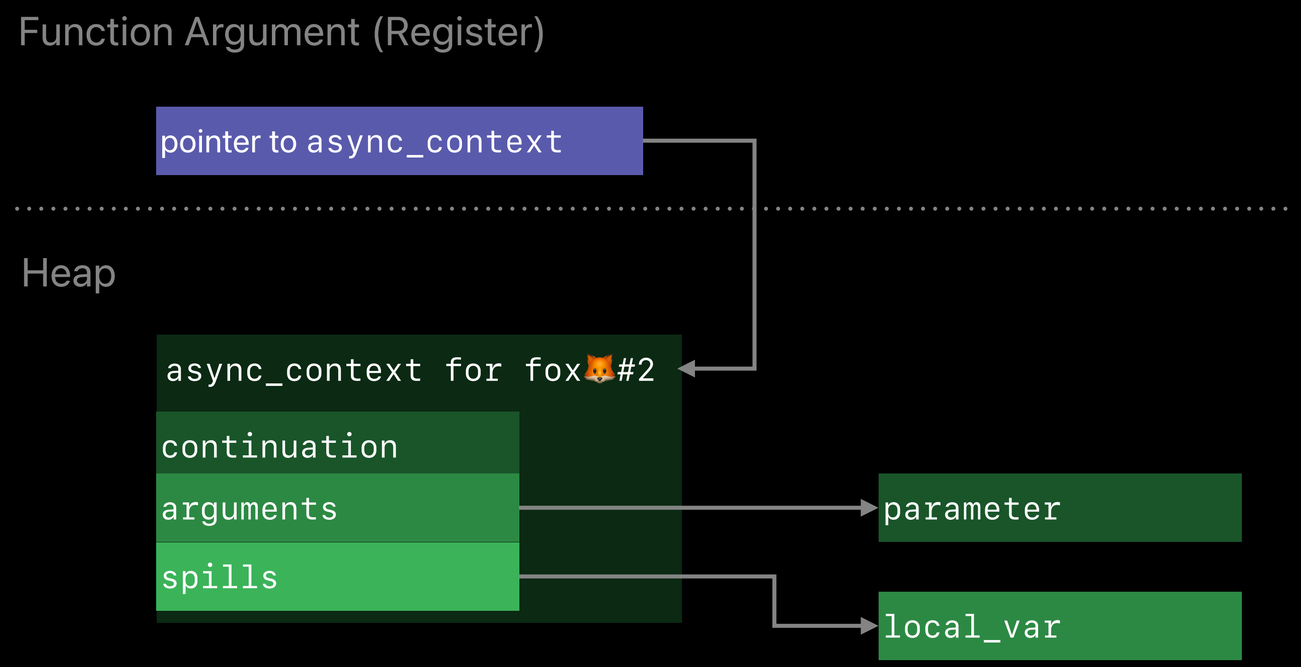
Variables

Heap Data Structure async_context

- Function Parameters
- Spilled Variables

Dedicated Register for Address of async_context

Guaranteed by Swift ABI




```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

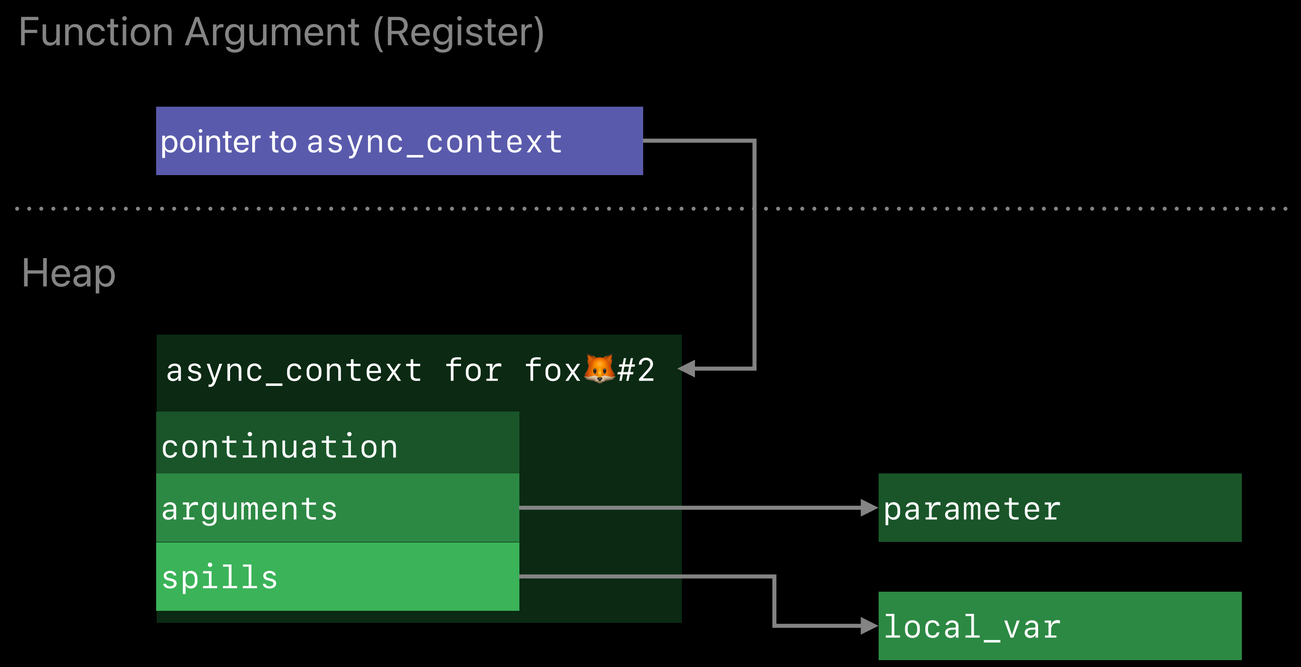
Variables

Heap Data Structure async_context

- Function Parameters
- Spilled Variables

Dedicated Register for Address of async_context

Guaranteed by Swift ABI




```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

Pointer to async_context Heap Object

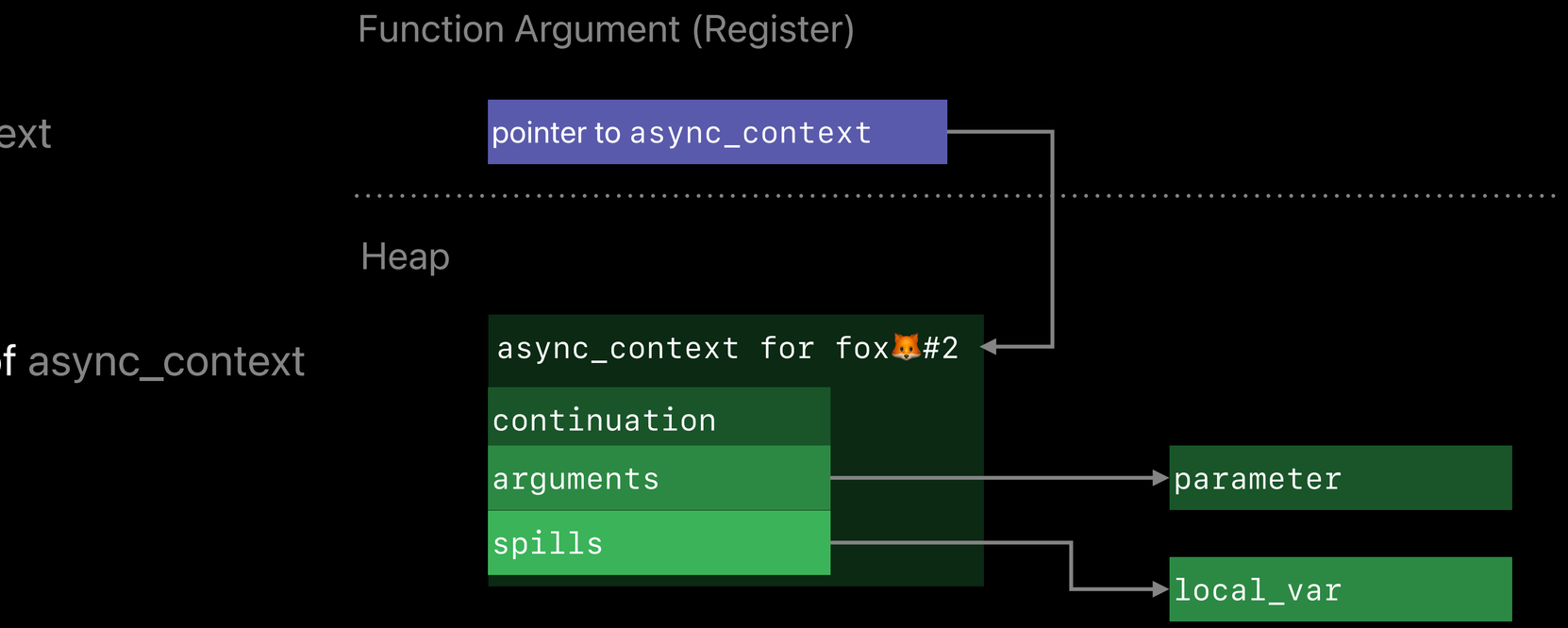
Variables

Heap Data Structure async_context

- Function Parameters
- Spilled Variables

Dedicated Register for Address of async_context

Guaranteed by Swift ABI




```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

Value of Register x22 at Entry of Function

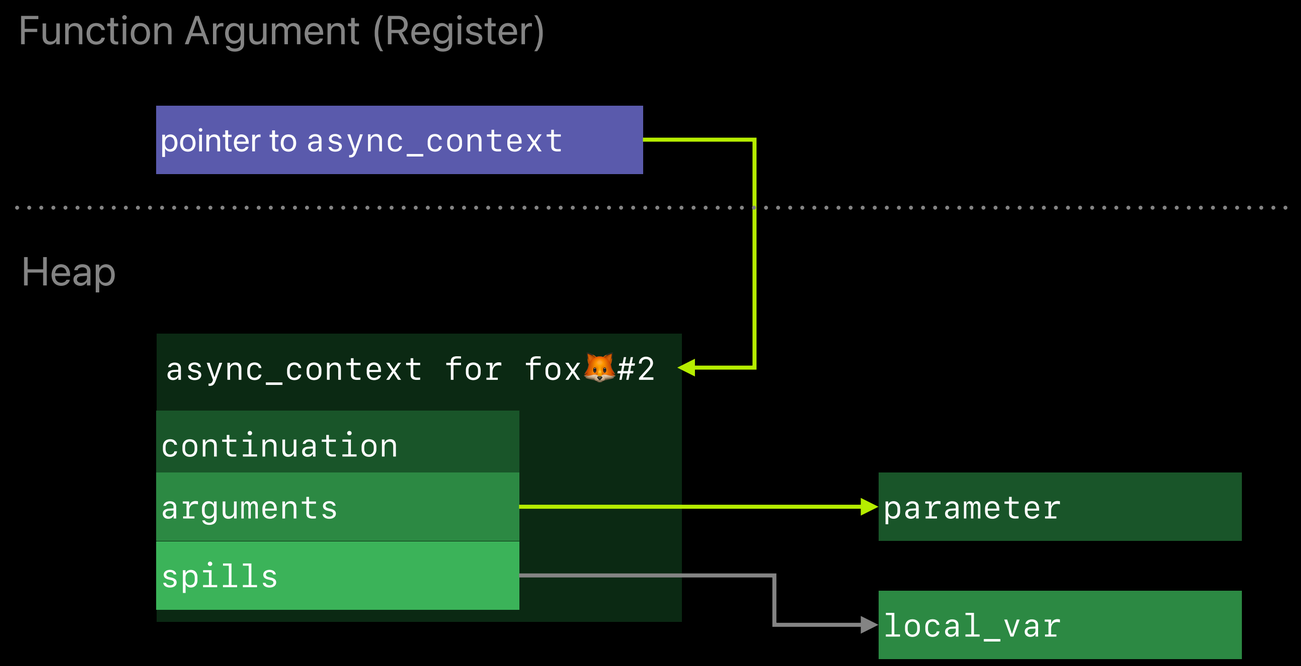
Variables

Heap Data Structure async_context

- Function Parameters
- Spilled Variables

Dedicated Register for Address of async_context

Guaranteed by Swift ABI




```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

Path to Variable in async_context*
*(x22+24)

*) This complex DWARF expression was generated by running llvm::salvageDebuginfo() until a fixed point was reached

Variables in Parent Frames

```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

Variables in Parent Frames

In synchronous code, debugger unwinds the stack to recover locals, restore registers

```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

Variables in Parent Frames

In synchronous code, debugger unwinds the stack to recover locals, restore registers

Async variables are described relative to `DW_OP_entry_value(DW_OP_reg22)`

```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

Variables in Parent Frames

In synchronous code, debugger unwinds the stack to recover locals, restore registers

Async variables are described relative to `DW_OP_entry_value(DW_OP_reg22)`

Works even in async parent frames:

- "Parent" "frames" are continuations
- Async continuations point to *beginning* of a new funclet
- Swift ABI dedicates register (`x22`) to pass `async_context`
- From this follows: the value of `x22` *must* be the address of `async_context`,
- Unwinder plugin can recover value from continuation's context

```
(lldb) image lookup -va $pc
...
Variable: id = {0x1000002c0}, name = "parameter", type = "String", valid ranges = <block>, location =
DW_OP_entry_value(DW_OP_reg22 x22), DW_OP_plus_uconst 0x18, DW_OP_deref, decl = main.swift:8
```

Summary

ABI, compiler, and debugger co-designed for async/await support

Summary

ABI, compiler, and debugger co-designed for async/await support

ABI

- Dedicated Register/Location for Context

Summary

ABI, compiler, and debugger co-designed for async/await support

ABI

- Dedicated Register/Location for Context

LLVM

- llvm::CoroCloner creates Entry Values and calls llvm::salvageDebugInfo()
- LiveDebugValues pass leaves Async Entry Values alone

Summary

ABI, compiler, and debugger co-designed for async/await support

ABI

- Dedicated Register/Location for Context

LLVM

- llvm::CoroCloner creates Entry Values and calls llvm::salvageDebugInfo()
- LiveDebugValues pass leaves Async Entry Values alone

LLDB

- Walk Continuations for Virtual Backtraces, and to simulate Stepping
- Unwinder recovers special Async Register

Summary

ABI, compiler, and debugger co-designed for async/await support

ABI

- Dedicated Register/Location for Context

LLVM

- llvm::CoroCloner creates Entry Values and calls llvm::salvageDebugInfo()
- LiveDebugValues pass leaves Async Entry Values alone

LLDB

- Walk Continuations for Virtual Backtraces, and to simulate Stepping
- Unwinder recovers special Async Register

Extensions being contributed back to LLVM now!

Summary

ABI, compiler, and debugger co-designed for async/await support

ABI

- Dedicated Register/Location for Context

LLVM

- llvm::CoroCloner creates Entry Values and calls llvm::salvageDebugInfo()
- LiveDebugValues pass leaves Async Entry Values alone

LLDB

- Walk Continuations for Virtual Backtraces, and to simulate Stepping
- Unwinder recovers special Async Register

Extensions being contributed back to LLVM now!