Async Functions in Swift

John McCall, Apple
Arnold Schwaighofer, Apple

Async functions

- Some operations spend a lot of time just waiting:
  - for a timer
  - for I/O to complete
  - for a server response
  - for a user action

Waiting on a server

```
func submitTurn() {
  let msg = gameState.formatAsJSON()

  let rawResponse = sendToServer(msg)

  gameState = GameState(json: rawResponse)
}
```

Waiting on a server

```
func submitTurn() {
  let msg = gameState.formatAsJSON()

  let rawResponse = sendToServer(msg)

  gameState = GameState(json: rawResponse)
}
```

Waiting on a server

```
func submitTurn() {
    let msg = gameState.formatAsJSON()

    let rawResponse = sendToServer(msg)

    gameState = GameState(json: rawResponse)
}
```

Threads are precious

- Programs often have a lot they can do while waiting
- Can create lots of threads, but that typically scales poorly:
  - C ABIs require a large, contiguous stack per thread
  - significant switching costs
  - relatively high creation cost
  - interacts poorly with use of threads for exclusion
- Lots of work done to lower these overheads, but threads remain precious

Thread abandonment

- Async functions are a different kind of function
- They can efficiently wait by giving up their thread:
  - occupy a thread normally when running
  - store state that’s live across a wait off of the thread’s stack
  - wait by returning and letting the thread do something else
  - resume by having their continuation called

Async functions

```
func submitTurn() async {
  let msg = gameState.formatAsJSON()

  let rawResponse = await sendToServer(msg)

  gameState = GameState(json: rawResponse)
}
```

Async functions

```
func submitTurn() async {
  let msg = gameState.formatAsJSON()

  let rawResponse = await sendToServer(msg)

  gameState = GameState(json: rawResponse)
}
```

```
func sendToServer(String) async -> String
```

Async functions

```
func submitTurn() async {
  let msg = gameState.formatAsJSON()

  let rawResponse = await sendToServer(msg)

  gameState = GameState(json: rawResponse)
}
```

```
func sendToServer(String) async -> String
```

Async functions

```
func submitTurn() async {
  let msg = gameState.formatAsJSON()

  let rawResponse = await sendToServer(msg)

  gameState = GameState(json: rawResponse)
}
```

```
func sendToServer(String) async -> String
```

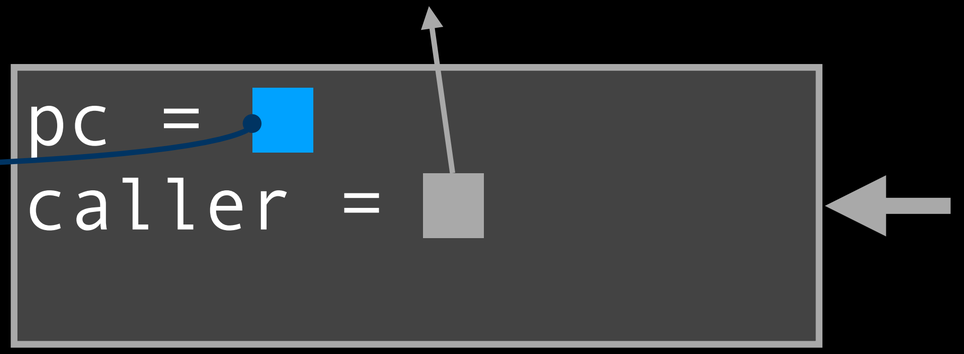
Async functions

```
func submitTurn() async {
  await sendToServer()
}
```

```
func sendToServer() async
```

Async functions

```
func submitTurn() async {
  await sendToServer()
}
```

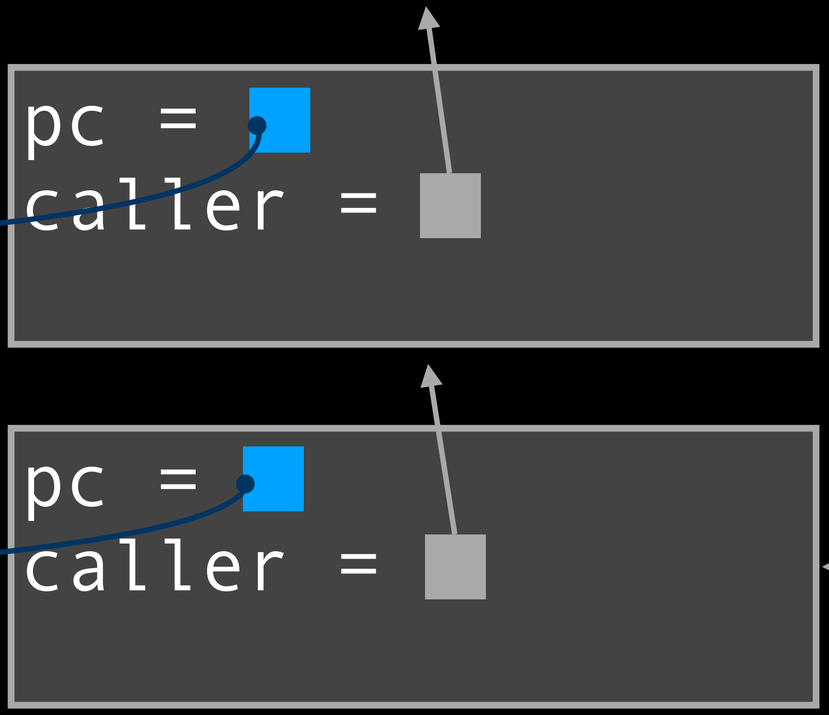



```
func sendToServer() async
```

Async functions

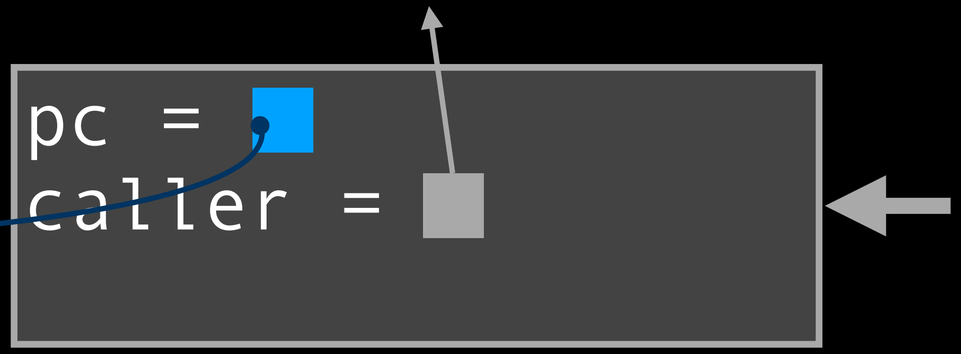
```
func submitTurn() async {
  await sendToServer()
}
```

```
func sendToServer() async
```



Async functions

```
func submitTurn() async {
  await sendToServer()
}
```




```
func sendToServer() async
```

Async functions

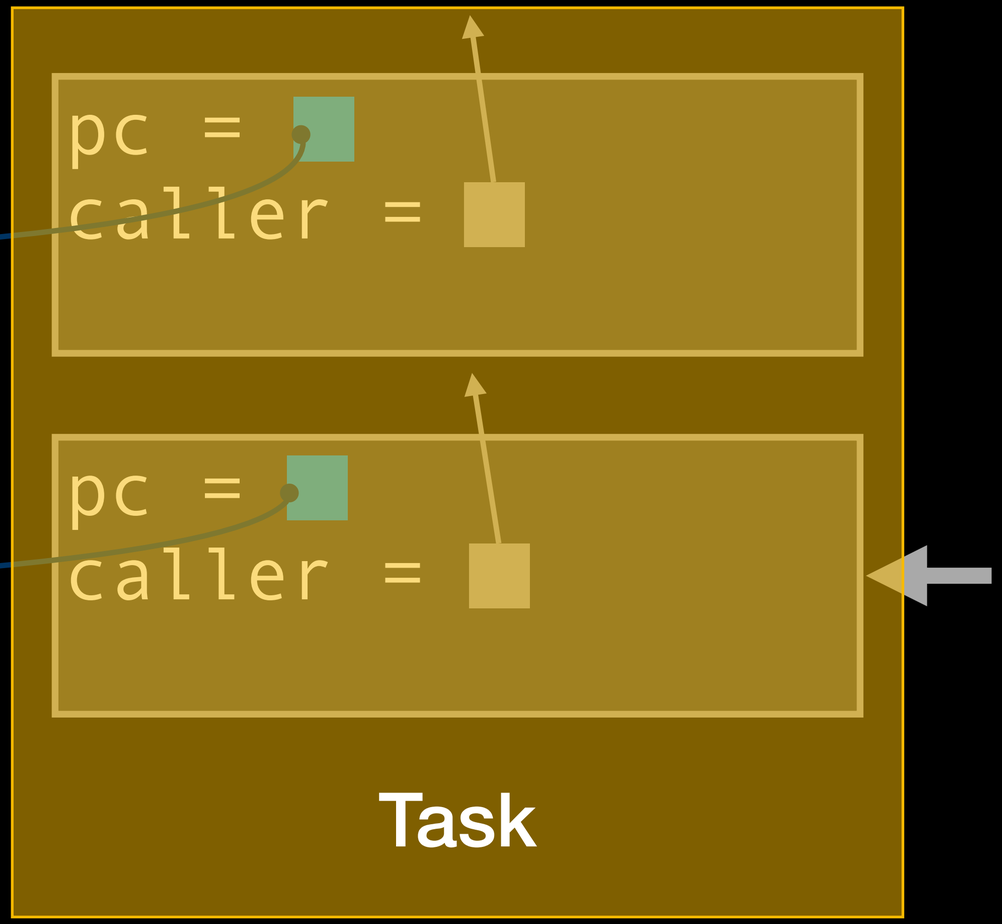
```
func submitTurn() async {
  await sendToServer()
}
```

```
func sendToServer() async
```

Async functions

```
func submitTurn() async {
  await sendToServer()
}
```

```
func sendToServer() async
```



Coroutines

Coroutines

- Gor Nishanov (LLVM 2016): “LLVM Coroutines”
  - https://www.youtube.com/watch?v=Ztr8QvMhqmQ
  - Focused on implementing the C++ coroutine feature
- John McCall (LLVM 2018): “Coroutine Representations and ABIs in LLVM”
  - https://www.youtube.com/watch?v=wyAbV8AM9PM
  - General background and some initial usage in Swift (accessors)

What’s a coroutine?

- Formally, a function that can suspend other than to initiate a call
  - e.g. a generator, which can yield a value, get resumed, and keep yielding more values

What’s a coroutine?

- Formally, a function that can suspend other than to initiate a call
  - e.g. a generator, which can yield a value, get resumed, and keep yielding more values
- In practice, strongly associated with the implementation technique of function splitting
  - People sometimes say “coroutine” for any function that will be split

Are async functions coroutines?

- Semantically, async functions just wait for other async calls to complete
  - could in principle be implemented by spawning / blocking threads
  - …but the point is to *not* block threads

Partial async functions

- Instead we use function splitting as an implementation technique
- Partial functions:
  - run between each potential waiting point
  - always tail call the next part to run
  - wait by just returning

Partial async functions

```
func submitTurn() async {
  let msg = …
  await sendToServer(msg)
  gameState = …
}
```

Partial async functions

```
func submitTurn() async {
  let msg = …
  await sendToServer(msg)
  gameState = …
}
```

```
func submitTurn() {
  let msg = …
  next = submitTurn_2
  tail call sendToServer(msg)
}
```

```
func submitTurn_2() {
  gameState = …
  tail call caller.next
}
```

```
func submitTurn() {
  let msg = …
  next = submitTurn_2
  tail call sendToServer(msg)
}
```

```
func sendToServer(msg) {
  …
  next = sendToServer_2
  tail call waitForResponse()
}
```

```
func sendToServer_2 {
  tail call caller.next
}
```

```
func submitTurn_2() {
  gameState = …
  tail call caller.next
}
```

Coroutine lowerings

- Central thesis of my talk from 2018:
  - There are many different use cases for coroutines
  - Different use cases want different low-level treatment
- Async functions are very different from accessors and generators
- Desirable implementation details are very different, too

Are async functions coroutines?

Thread’s perspective

- To a thread, an async function is a coroutine
- The thread executes a succession of partial functions
- At any time, one might return, having logically suspended the task

Task’s perspective

- To a task, an async function is an ordinary routine
- Async functions call each other and wait for each other to return
  - Swift enforces this call/return structure by default
  - Forking off a new task is an explicit operation

Familiar ideas from routines

- Internal representation in the compiler
- Compiler optimization
- Stack allocation
- Backtraces

Async inlining in SIL

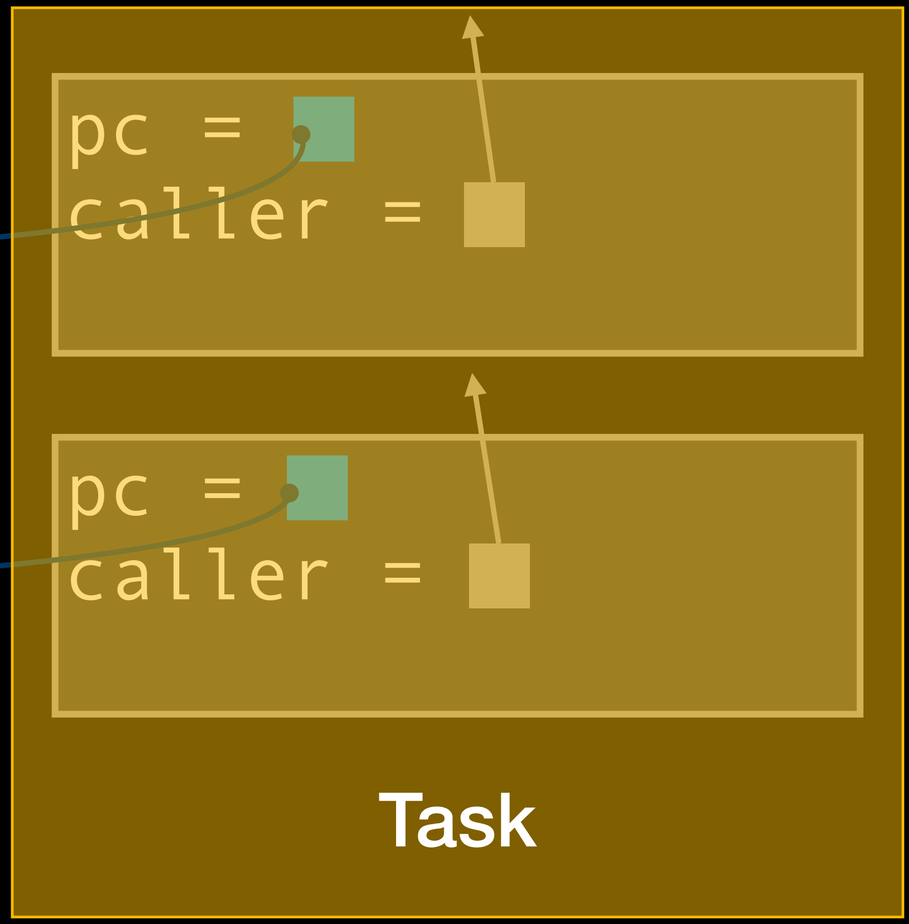
```
sil @submitTurn : $@async () -> @error Error {
bb0:
  …
  %10 = function_ref @sendToServer : $@async
(@guaranteed String) -> (@owned String, @error Error)
  try_apply %10(%8) : $@async (@guaranteed String) ->
(@owned String, @error Error), normal bb1, error bb2
  …
}

sil @sendToServer : $@async (@guaranteed String) ->
@owned String {
  …
```

Stack allocation

```
func submitTurn() async {
  await sendToServer()
}
```

```
func sendToServer() async
```



Context allocation

```
func submitTurn() async {




  await sendToServer()

}
```

Context allocation

```
func submitTurn(i8* %frame) async {




  await sendToServer()

}
```

Context allocation

```
func submitTurn(i8* %frame) async {
  %size = load i32* sendToServer.frameSize



  await sendToServer()

}
```

Context allocation

```
func submitTurn(i8* %frame) async {
  %size = load i32* sendToServer.frameSize
  %calleeFrame = swift_task_alloc(%size)


  await sendToServer()
  swift_task_dealloc(%calleeFrame)
}
```

Context allocation

```
func submitTurn(i8* %frame) async {
  %size = load i32* sendToServer.frameSize
  %calleeFrame = swift_task_alloc(%size)
  store %frame, %calleeFrame.caller
  store submitTurn_2, %calleeFrame.continuation
  await sendToServer()
  swift_task_dealloc(%calleeFrame)
}
```

Context allocation

```
func submitTurn(i8* %frame) async {
  %size = load i32* sendToServer.frameSize
  %calleeFrame = swift_task_alloc(%size)
  store %frame, %calleeFrame.caller
  store submitTurn_2, %calleeFrame.continuation
  await sendToServer(%calleeFrame)
  swift_task_dealloc(%calleeFrame)
}
```

Trade-offs

- Typical of Swift's general approach:
  - works well across ABI / polymorphic boundaries
  - allows flexibility, accepts performance overhead, tries to mitigate
  - static optimization opportunities (e.g. to co-allocate frames)
  - success of static optimization could theoretically be enforced
  - reliably preserves information dynamically for tooling

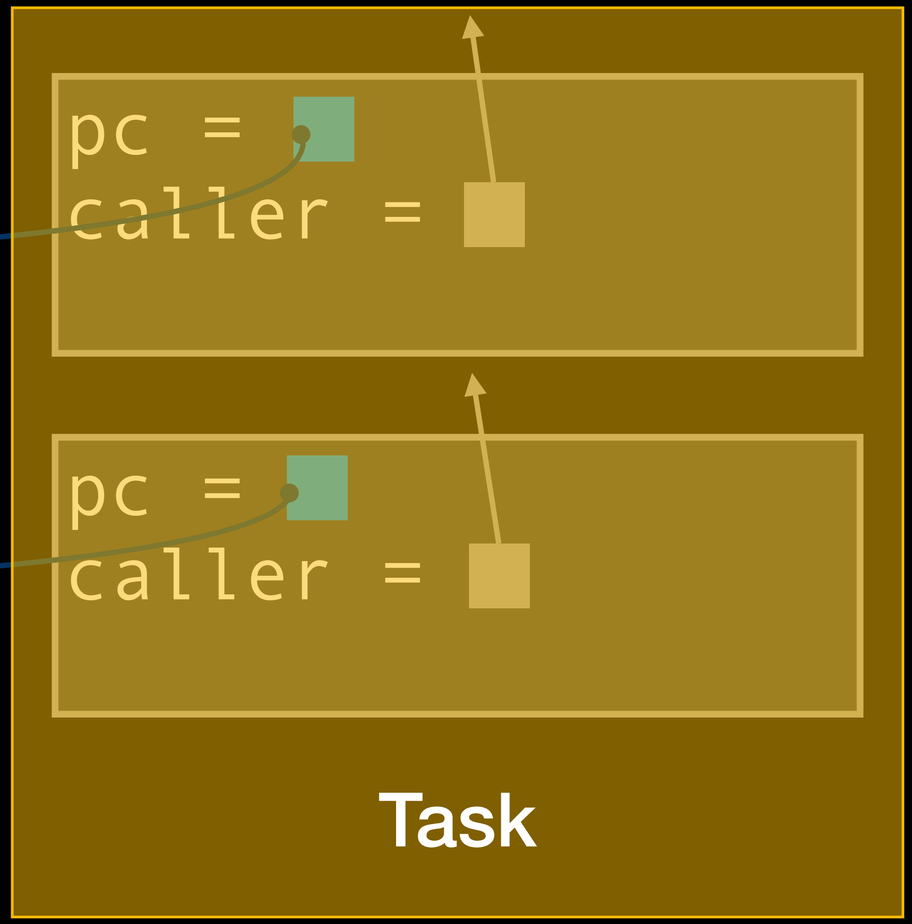
Other languages

- Most languages with async functions make them less “structured”
- Usually async functions return explicit *promises* that can be awaited
- Promise features vary, but often:
  - can have zero awaiters
  - can have multiple awaiters
  - can be concurrently awaited
  - can be forwarded around
  - can be produced in ways besides async functions

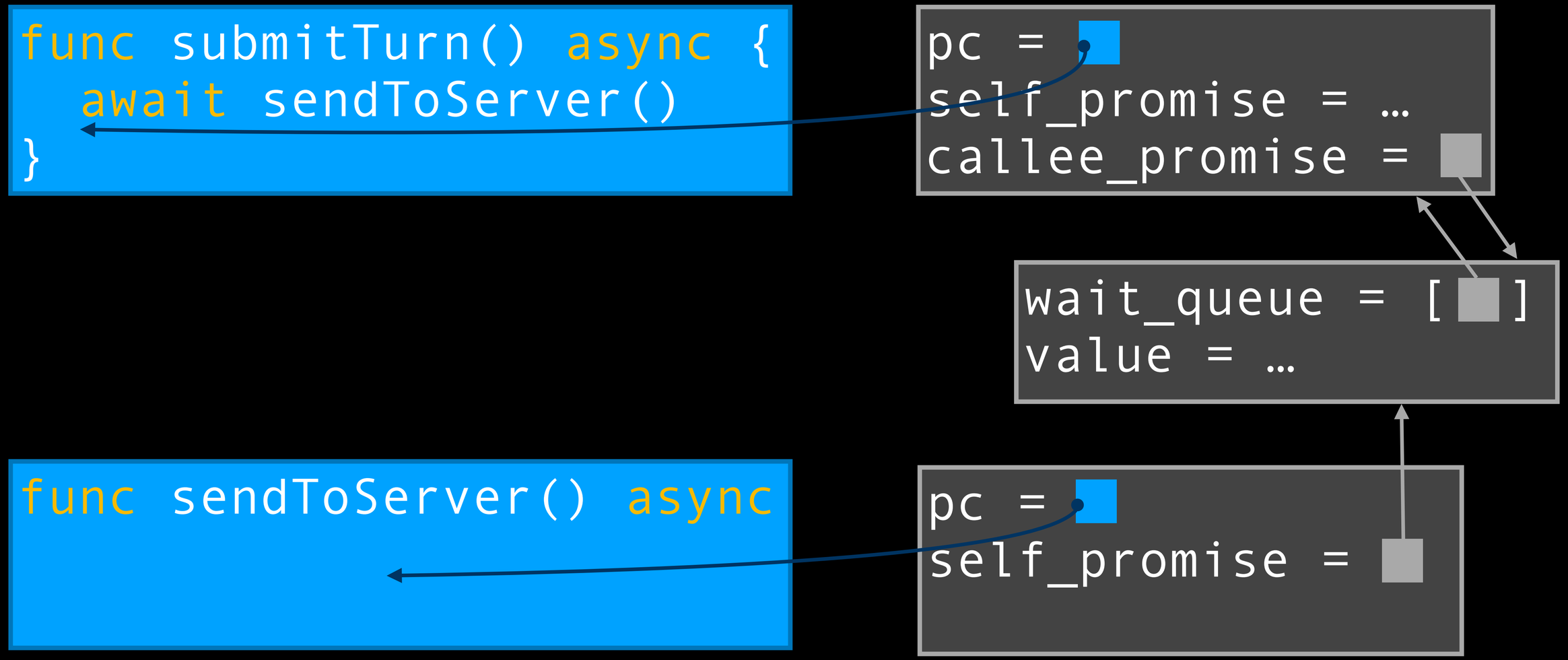
Async context allocation

```
func submitTurn() async {
  await sendToServer()
}
```

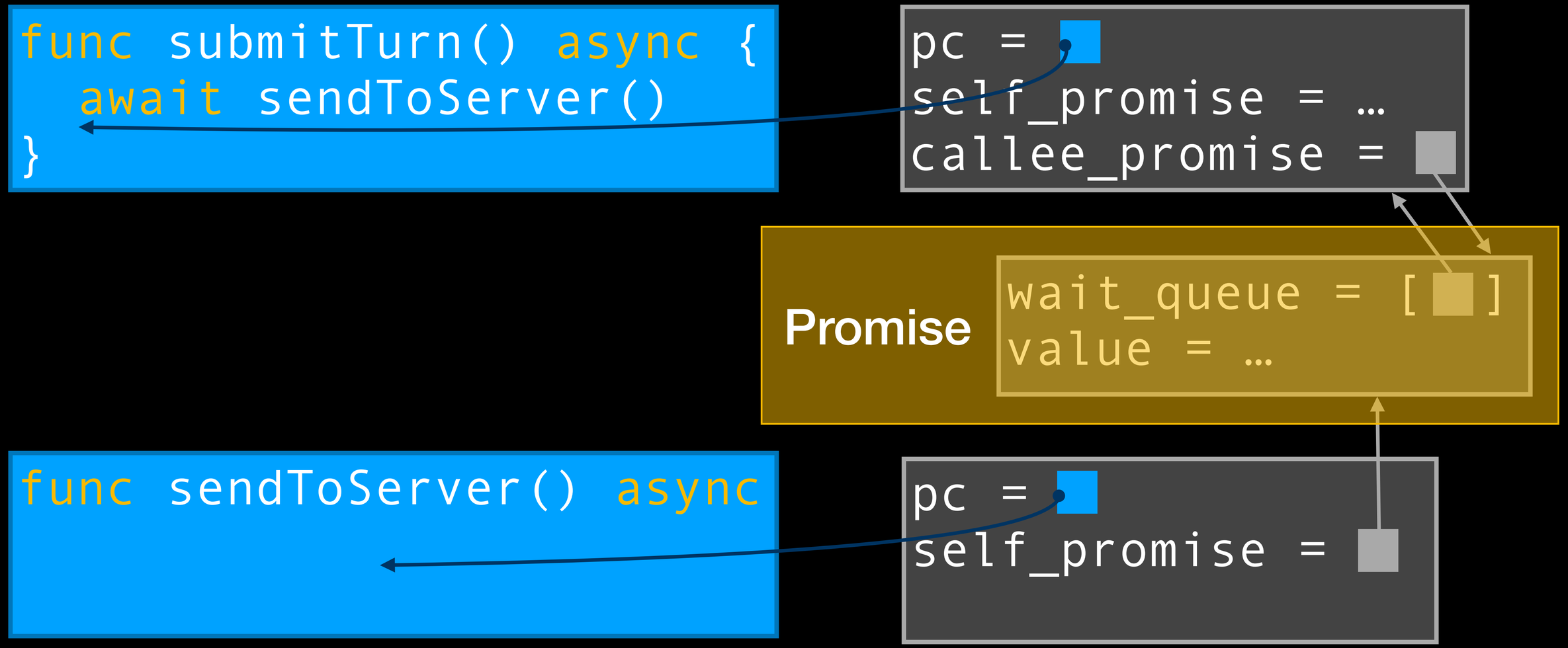
```
func sendToServer() async
```



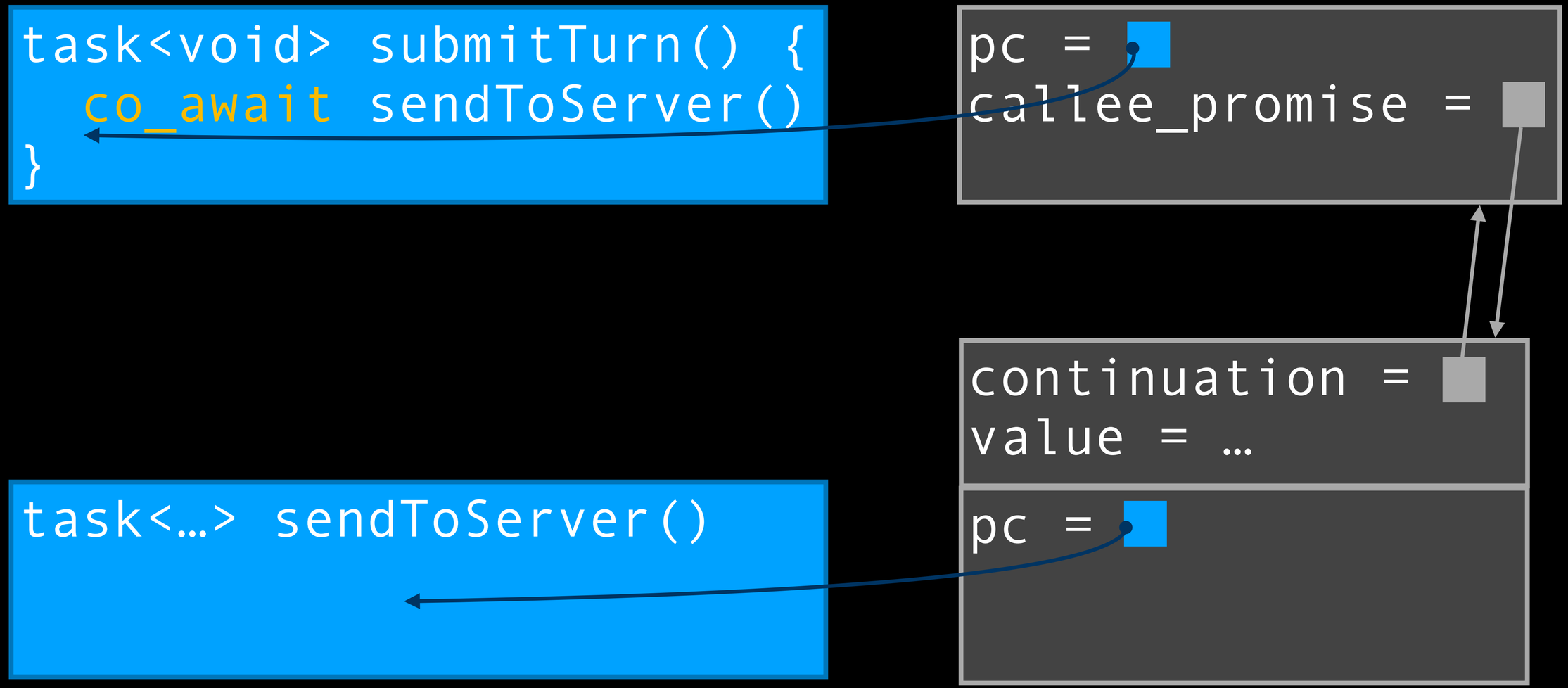
Async contexts with promises



Async contexts with promises



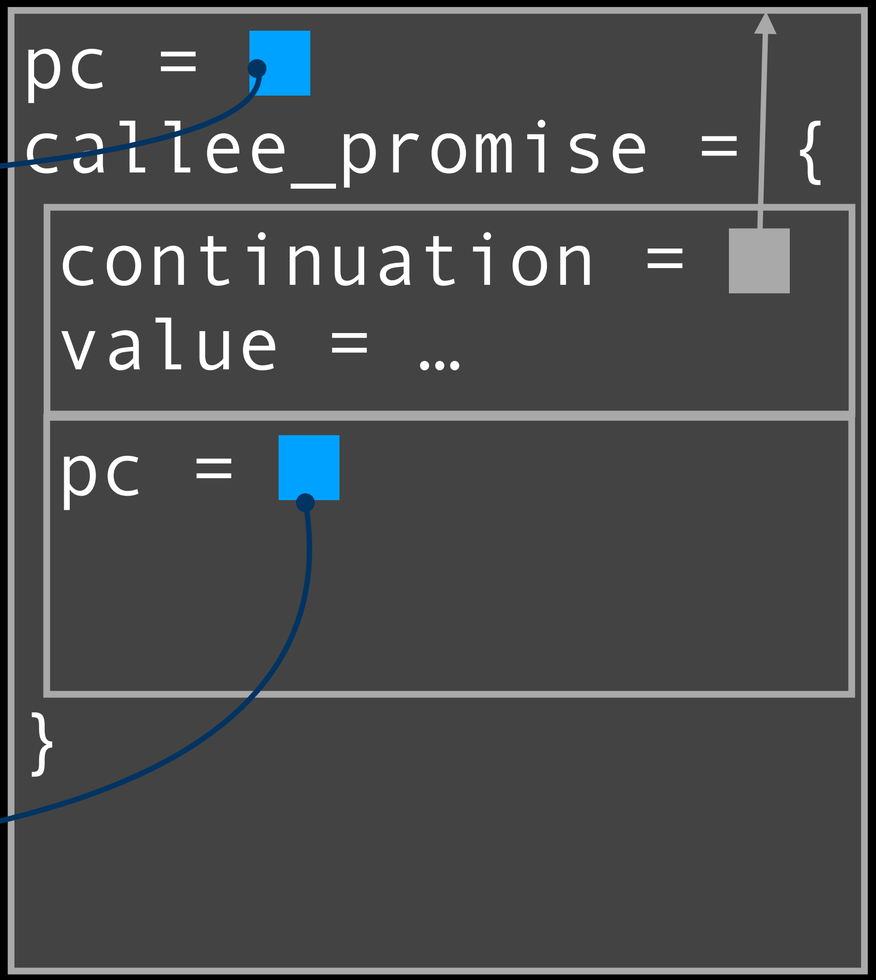
C++ promises (P1056, `std::task`)



Rust promises (futures)

```
async fn submitTurn() {
  sendToServer().await
}
```

```
async fn sendToServer()
```



Feature Set

- To support async await style calls

Feature Set

- To support async await style calls
- Split source level functions into fragments

Feature Set

- To support async await style calls
- Split source level functions into fragments
- Mandatory tail call optimization support

Feature Set

- To support async await style calls
- Split source level functions into fragments
- Mandatory tail call optimization support
- Efficient parameter passing ABI

Feature Set

- To support async await style calls
- Split source level functions into fragments
- Mandatory tail call optimization support
- Efficient parameter passing ABI
- Tooling support (debugger, backtrace)

Fragments

- Calling an asynchronous function may suspend execution

```
void @function1(…) {
   call void @print()

   // suspend point
   call void @function2(…)

   call void @print2()
}
```

Fragments

- Calling an asynchronous function may suspend execution

```
void @function1(…) {
   call void @print()

   // suspend point
   call void @function2(…)

   call void @print2()
}
```

```
void @function1() {
   call void @print()

   %async_ctx = call @swift_task_alloc(…)
   %async_ctx.ResumeInParent = @function1_fragment1
```

```
void @function1_fragment1(…) {

   call @print2()
}
```

Fragments

- Calling an asynchronous function may suspend execution

```
void @function1(…) {
   call void @print()

   // suspend point
   call void @function2(…)

   call void @print2()
}
```

```
void @function1() {
   call void @print()

   %async_ctx = call @swift_task_alloc(…)
   %async_ctx.ResumeInParent = @function1_fragment1

   call @function2(%async_ctx)
}

void @function1_fragment1(…) {

   call @print2()
}
```

Tail Call Support

- Fragments tail call each other

```
swiftailcc void @fragment(<num_args>) {

   musttail call swiftailcc void @fragment2(<num_args + N>)
   ret void
}
```

Tail Call Support

- Fragments tail call each other
- Pass arbitrary number of arguments as efficiently as a regular call

```
swiftailcc void @fragment(<num_args>) {

   musttail call swiftailcc void @fragment2(<num_args + N>)
   ret void
}
```

Tail Call Support

- Fragments tail call each other
- Pass arbitrary number of arguments as efficiently as a regular call
  - `swiftailcc` is a callee pops convention

```
swiftailcc void @fragment(<num_args>) {

   musttail call swiftailcc void @fragment2(<num_args + N>)
   ret void
}
```

Tail Call Support

- Fragments tail call each other
- Pass arbitrary number of arguments as efficiently as a regular call
  - `swiftailcc` is a callee pops convention
- Tail call has to reuse stack frame

```
swiftailcc void @fragment(<num_args>) {

   musttail call swiftailcc void @fragment2(<num_args + N>)
   ret void
}
```

Tail Call Support

- Fragments tail call each other
- Pass arbitrary number of arguments as efficiently as a regular call
  - `swiftailcc` is a callee pops convention
- Tail call has to reuse stack frame
  - Tail call optimization is mandatory / must be preserved by optimizations (`musttail`)

```
swiftailcc void @fragment(<num_args>) {

   musttail call swiftailcc void @fragment2(<num_args + N>)
   ret void
}
```

Tooling Support

- Chain of async context frames form logical task stack

```
struct Async_Context {
  Async_Context *Parent;
  void          (*ResumeInParent)();
}
```

Tooling Support

- Chain of async context frames form logical task stack

```
struct Async_Context {
  Async_Context *Parent;
  void         (*ResumeInParent)();
}
```

```
struct Async_Context {
  Async_Context *Parent;
  void         (*ResumeInParent)();
}
```

Tooling Support

- Chain of async context frames form logical task stack

```
struct Async_Context {
  Async_Context *Parent;
  void          (*ResumeInParent)();
}
```

- Support debugger and backtrace

Tooling Support

- Chain of async context frames form logical task stack

```
struct Async_Context {
  Async_Context *Parent;
  void          (*ResumeInParent)();
}
```

- Support debugger and backtrace
  - Identify async stack frame: `swiftasync` parameter (reserved register)

Tooling Support

- Chain of async context frames form logical task stack

```
struct Async_Context {
  Async_Context *Parent;
  void          (*ResumeInParent)();
}
```

- Support debugger and backtrace
  - Identify async stack frame: `swiftasync` parameter (reserved register)

```
swifttailcc void @function1(i8* swiftasync %async_ctxt) {
}
```

Tooling Support

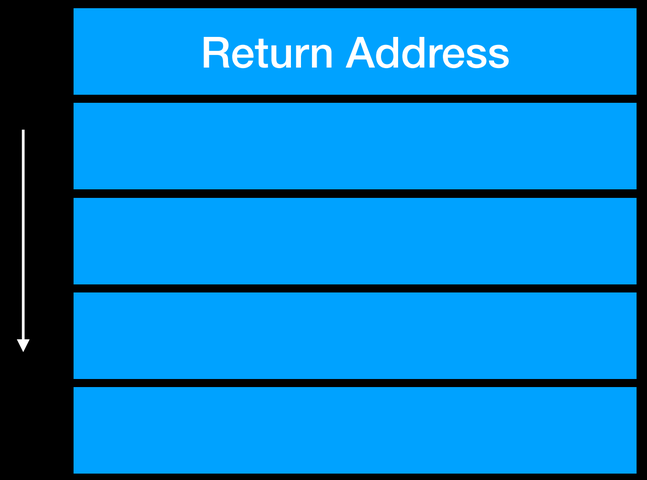
- Identify “extended” stack frame

```
swifttailcc void @function1_fragment0(i8* swiftasync %async_ctxt) {
}
```

Tooling Support

- Identify “extended” stack frame

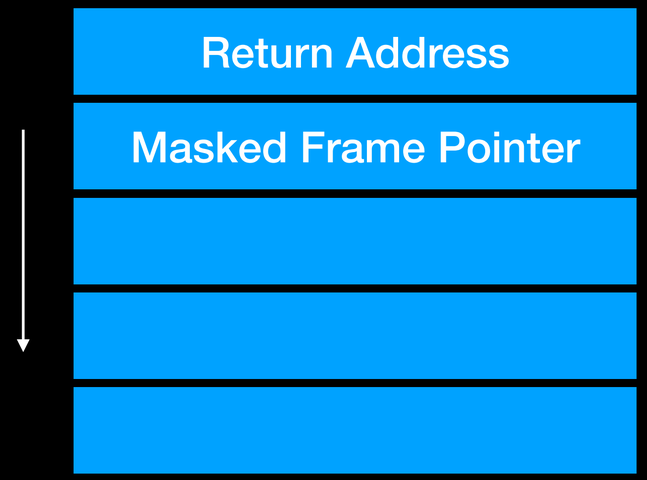
```
swifttailcc void @function1_fragment0(i8* swiftasync %async_ctxt) {
}
```



Tooling Support

- Identify “extended” stack frame

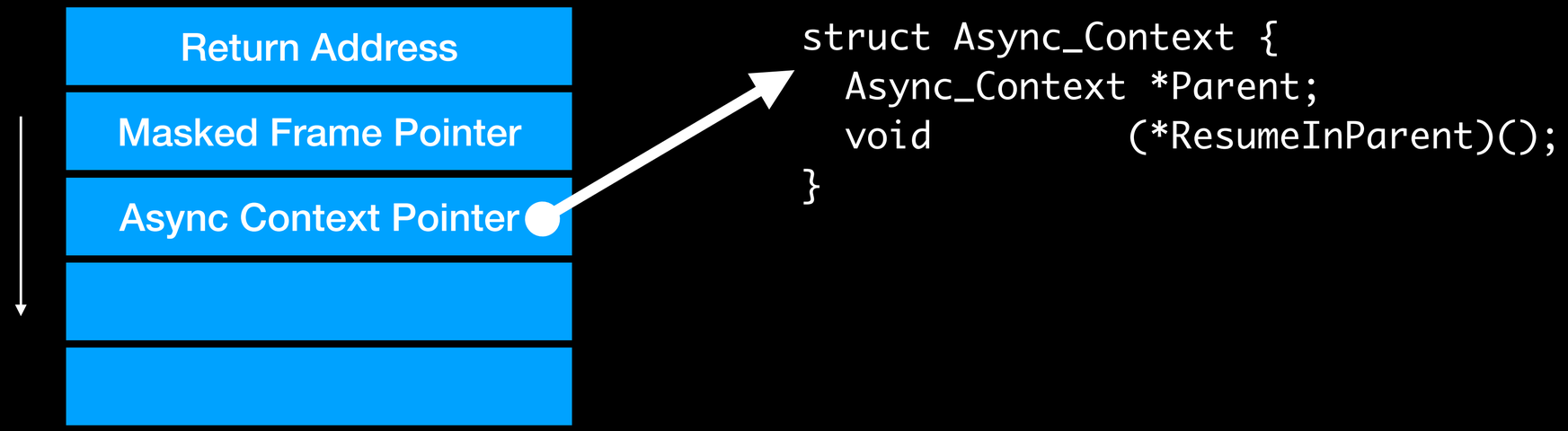
```
swifttailcc void @function1_fragment0(i8* swiftasync %async_ctxt) {
}
```



Tooling Support

- Identify “extended” stack frame

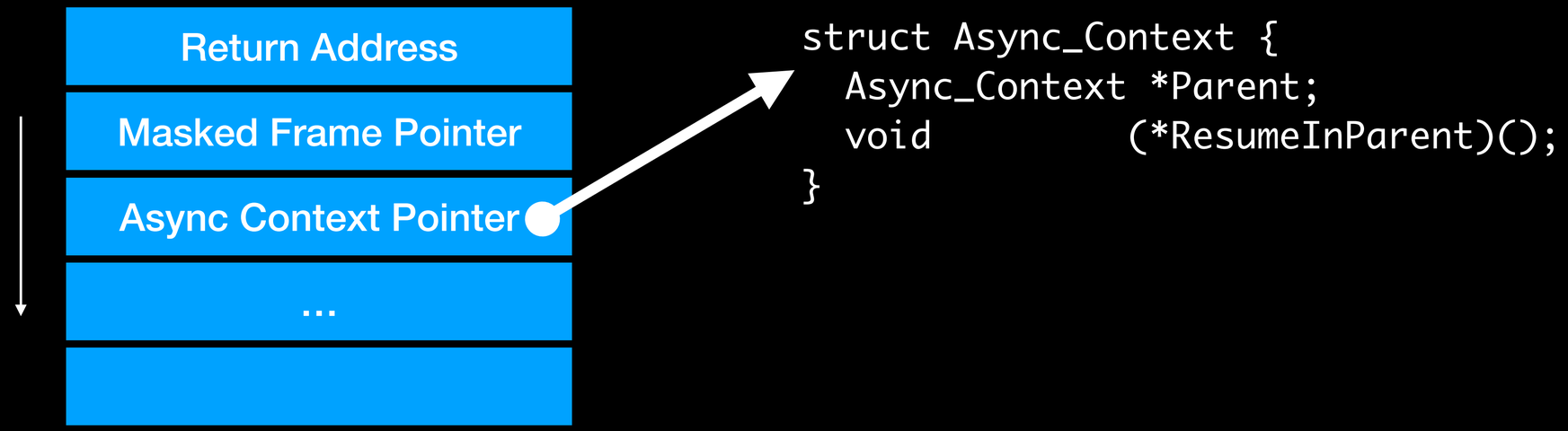
```
swifttailcc void @function1_fragment0(i8* swiftasync %async_ctxt) {
}
```



Tooling Support

- Identify “extended” stack frame

```
swifttailcc void @function1_fragment0(i8* swiftasync %async_ctxt) {
}
```



Live Values

- Async Context contains frame for values live across fragments

```
void @function1() {
  call @print()

  %val =

  call @function2() // suspend point

  call @print2(%val)
}
```

Live Values

- Async Context contains frame for values live across fragments

```
void @function1() {
  call @print()

  %val =

  call @function2() // suspend point

  call @print2(%val)
}
```

Live Values

- Async Context contains frame for value live across fragment

```
void @function1() {
  call @print()

  %val =

  call @function2() // suspend point

  call @print2(%val)
}
```

```
void @function1(i8* %async_ctx) {
   call void @print()

   %val =
   %async_context->frame.val_storage = %val

   call @function2(…)
}

 void @function1_fragment1(i8* %async_ctx) {

    %val_reload = %async_ctx->frame.val_storage

    call @print2(%val_reload)
 }
```

Live Values

- Async Context contains frame for value live across fragment

```
void @function1() {
  call @print()

  %val =

  call @function2() // suspend point

  call @print2(%val)
}
```

```
void @function1(i8* %async_ctx) {
   call void @print()

   %val =
   %async_context->frame.val_storage = %val

   call @function2(…)
}

 void @function1_fragment1(i8* %async_ctx) {

    %val_reload = %async_ctx->frame.val_storage

    call @print2(%val_reload)
 }
```

Live Values

- Async Context contains frame for value live across fragment

```
void @function1(i8* %async_ctx) {
   call void @print()

   %val =
   %async_context->frame.val_storage = %val

   call @function2(…)
}


 void @function1_fragment1(i8* %async_ctx) {

    %val_reload = %async_ctx->frame.val_storage

    call @print2(%val_reload)
 }
```

Live Values

- Async Context contains frame for value live across fragment

```
void @function1(i8* %async_ctx) {
   call void @print()

   %val =
   %async_context->frame.val_storage = %val

   call @function2(…)
}


 void @function1_fragment1(i8* %async_ctx) {

   %val_reload = %async_ctx->frame.val_storage

   call @print2(%val_reload)
 }
```

```
struct Async_Context {
  Async_Context *Parent;
  Void         (*ResumeInParent)();

  struct Async_Frame {
     int64_t spilledVar0;
     char    spilledAlloca[16];
     ...
  } frame;
}
```

Async Context Size

- Caller needs to allocate storage for call (across ABI boundaries)

```
%size_of_context = ???
%async_ctxt = swift_task_alloc(%size_of_context)
musttail call swifttailcc @function1(i8* swiftasync %async_ctxt, ...)
```

Async Context Size

- Caller needs to allocate storage for call (across ABI boundaries)

```
%size_of_context = ???
%async_ctxt = swift_task_alloc(%size_of_context)
musttail call swifttailcc @function1(i8* swiftasync %async_ctxt, ...)
```

- Async function pointer

```
@function1_afp = <callee, callee_context_size>

%size_of_context = load (gep @function1_afp, 1)
```

Async Context Size

- Caller needs to allocate storage for call (across ABI boundaries)

```
%size_of_context = ???
%async_ctxt = swift_task_alloc(%size_of_context)
musttail call swifttailcc @function1(i8* swiftasync %async_ctxt, ...)
```

- Async function pointer

```
@function1_afp = <callee, callee_context_size>

%size_of_context = load (gep @function1_afp, 1)
```

- Code deals in async function pointer (vtable of afp)

Async Context Size

- Caller needs to allocate storage for call (across ABI boundaries)

```
%size_of_context = ???
%async_ctxt = swift_task_alloc(%size_of_context)
musttail call swifttailcc @function1(i8* swiftasync %async_ctxt, ...)
```

- Async function pointer

```
@function1_afp = <callee, callee_context_size>

%size_of_context = load (gep @function1_afp, 1)
```

- Code deals in async function pointer (vtable of afp)

- Final context size computed by coroutine lowering pass

LLVM Async Coro Intrinsics

- Basic Skeleton
- Suspend Point

Basic Skeleton

- coro.async.id
  - Initial context size (reserved for frontend)

```
@function1_afp = <i32, i32> <@function1, 16>

swiftailcc void @function1(i8* swiftasync %ctx, …) {

  %id = call @coro.id.async(i32 16
```

Basic Skeleton

- coro.async.id
  - Initial context size (reserved for frontend)
  - Initial alignment of the context

```
@function1_afp = <i32, i32> <@function1, 16>

swiftailcc void @function1(i8* swiftasync %ctx, …) {

  %id = call @coro.id.async(i32 16, i32 16,
```

Basic Skeleton

- coro.async.id
  - Initial context size (reserved for frontend)
  - Initial alignment of the context
  - Context parameter index

```
@function1_afp = <i32, i32> <@function1, 16>

swiftailcc void @function1(i8* swiftasync %ctx, …) {

  %id = call @coro.id.async(i32 16, i32 16, i32 0,
```

Basic Skeleton

- coro.async.id
  - Initial context size (reserved for frontend)
  - Initial alignment of the context
  - Context parameter index
  - Async function pointer reference

```
@function1_afp = <i32, i32> <@function1, 16>

swiftailcc void @function1(i8* swiftasync %ctx, …) {

  %id = call @coro.id.async(i32 16, i32 16, i32 0,
                            i8* @function1_afp)
```

Basic Skeleton

- coro.async.id
  - Initial context size (reserved for frontend)
  - Initial alignment of the context
  - Context parameter index
  - Async function pointer reference

```
@function1_afp = <i32, i32> <@function1, 16>

swiftailcc void @function1(i8* swiftasync %ctx, …) {

  %id = call @coro.id.async(i32 16, i32 16, i32 0,
                            i8* @function1_afp)
  %hdl = call @coro.begin(%id)
```

Basic Skeleton

- coro.async.id
  - Initial context size (reserved for frontend)
  - Initial alignment of the context
  - Context parameter index
  - Async function pointer reference

```
@function1_afp = <i32, i32> <@function1, 16>

swiftailcc void @function1(i8* swiftasync %ctx, …) {

  %id = call @coro.id.async(i32 16, i32 16, i32 0,
                            i8* @function1_afp)
  %hdl = call @coro.begin(%id)


  tail call %ctx->ResumeInParent(%ctx, %results…)
```

Basic Skeleton

- coro.async.id
  - Initial context size (reserved for frontend)
  - Initial alignment of the context
  - Context parameter index
  - Async function pointer reference
- coro.end.async
  - Optional additional parameters
    - (function to tail call)

```
@function1_afp = <i32, i32> <@function1, 16>

swiftailcc void @function1(i8* swiftasync %ctx, …) {

  %id = call @coro.id.async(i32 16, i32 16, i32 0,
                            i8* @function1_afp)
  %hdl = call @coro.begin(%id)


  tail call %ctx->ResumeInParent(%ctx, %results…)
  call @coro.end.async(%hdl, …)
  unreachable
}
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {
  …

  %res = call {i8*, …res} @coro.suspend.async(
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended
  - Async context parameter index

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {
  …

  %res = call {i8*, …res} @coro.suspend.async(i32 0,
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended
  - Async context parameter index
  - Reference to the resume fragment place holder

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %resume = call i8* coro.async.resume()
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = %resume

  %res = call {i8*, …res} @coro.suspend.async(i32 0,
                                 i8* %resume,
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended
  - Async context parameter index
  - Reference to the resume fragment place holder
  - How to restore the current context from the async context passed

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %resume = call i8* coro.async.resume()
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = %resume
  %callee_ctx->Parent = %ctx
  %res = call {i8*, …res} @coro.suspend.async(i32 0,
                                i8* %resume,
                                i8*(i8*)* @project_ctxt,
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended
  - Async context parameter index
  - Reference to the resume fragment place holder
  - How to restore the current context from the async context passed
  - Forwarding function that models the tail call to the callee

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %resume = call i8* coro.async.resume()
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = %resume
  %callee_ctx->Parent = %ctx
  %res = call {i8*, …res} @coro.suspend.async(i32 0,
                                i8* %resume,
                                i8*(i8*)* @project_ctxt,
                                void (i8*, i8*, …) @forwarder,
                                i8* @function2,
                                i8* %callee_ctx, …args)
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended
  - Async context parameter index
  - Reference to the resume fragment place holder
  - How to restore the current context from the async context passed
  - Forwarding function that models the tail call to the callee

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %resume = call i8* coro.async.resume()
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = %resume
  %callee_ctx->Parent = %ctx
  %res = call {i8*, …res} @coro.suspend.async(i32 0,
                          i8* %resume,
                          i8*(i8*)* @project_ctxt,
                          void (i8*, i8*, …) @forwarder,
                          i8* @function2,
                          i8* %callee_ctx, …args)
```

```
swiftailcc void @forwarder(i8* %fun, i8* %ctx, …args) {
   musttail call swiftailcc %fun(i8* %ctx, …args)
   ret void
}
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended
  - Async context parameter index
  - Reference to the resume fragment place holder
  - How to restore the current context from the async context passed
  - Forwarding function that models the tail call to the callee

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %resume = call i8* coro.async.resume()
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = %resume
  %callee_ctx->Parent = %ctx
  %res = call {i8*, …res} @coro.suspend.async(i32 0,
                                i8* %resume,
                                i8*(i8*)* @project_ctxt,
                                void (i8*, i8*, …) @forwarder,
                                i8* @function2,
                                i8* %callee_ctx, …args)
```

Suspend point

- coro.suspend.async
  - Models call which could be suspended
  - Async context parameter index
  - Reference to the resume fragment place holder
  - How to restore the current context from the async context passed
  - Forwarding function that models the tail call to the callee
  - Extract the result(s)

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %resume = call i8* coro.async.resume()
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = %resume
  %callee_ctx->Parent = %ctx
  %res = call {i8*, …res} @coro.suspend.async(i32 0,
                                i8* %resume,
                                i8*(i8*)* @project_ctxt,
                                void (i8*, i8*, …) @forwarder,
                                i8* @function2,
                                i8* %callee_ctx, …args)

  call @swift_task_dealloc(%callee_ctxt)
  %result = extract_value %res, 1
  call @use(i64 %result)
…
}
```

Coroutine Lowering

- Splits functions

Coroutine Lowering

- Splits functions

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = @function1_fragment2
  %callee_ctx->Parent = %ctx
  musttail call swifttailcc @function2(
                                    i8* %callee_ctxt)
  ret void
}

swiftailcc void @function1_fragment2(
  i8* swiftasync %callee_txt, i64 %result) {

  %ctx = call @project_ctxt(%callee_ctxt)
  call @swift_task_dealloc(%callee_ctxt)
  call @use(i64 %result)
  …
}
```

Coroutine Lowering

- Splits functions
- Replaces resume function place holder

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = @function1_fragment2
  %callee_ctx->Parent = %ctx
  musttail call swifttailcc @function2(
                                    i8* %callee_ctxt)
  ret void
}

swiftailcc void @function1_fragment2(
  i8* swiftasync %callee_txt, i64 %result) {

  %ctx = call @project_ctxt(%callee_ctxt)
  call @swift_task_dealloc(%callee_ctxt)
  call @use(i64 %result)
  …
}
```

Coroutine Lowering

- Splits functions
- Replaces resume function place holder
- Handles live values / addresses

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = @function1_fragment2
  %callee_ctx->Parent = %ctx
  musttail call swifttailcc @function2(
                                    i8* %callee_ctxt)
  ret void
}

swiftailcc void @function1_fragment2(
  i8* swiftasync %callee_txt, i64 %result) {

  %ctx = call @project_ctxt(%callee_ctxt)
  call @swift_task_dealloc(%callee_ctxt)
  call @use(i64 %result)
  %reload = %ctx->frame.val
}
```

Coroutine Lowering

- Splits functions
- Replaces resume function place holder
- Handles live values / addresses
- Fixup of the result value(s)

```
swiftailcc void @function1(i8* swiftasync %ctx, …) {

  …
  %callee_ctx = call @swift_task_alloc(…)
  %callee_ctx->ResumeInParent = @function1_fragment2
  %callee_ctx->Parent = %ctx
  musttail call swifttailcc @function2(
                                    i8* %callee_ctxt)
  ret void
}

swiftailcc void @function1_fragment2(
  i8* swiftasync %callee_txt, i64 %result) {

  %ctx = call @project_ctxt(%callee_ctxt)
  call @swift_task_dealloc(%callee_ctxt)
  call @use(i64 %result)
  …
}
```

Future Improvements

Future Improvements

- Merge async context stack allocations

```
…
  %callee_ctx = call @swift_task_alloc(16)
  call @coro.async.suspend(…)
  call @swift_task_dealloc(%callee_ctxt)

  %callee_ctx2 = call @swift_task_alloc(32)
  call @coro.async.suspend(…)
  call @swift_task_dealloc(%callee_ctxt2)
```

Future Improvements

- Merge async context stack allocations

```
…
%callee_ctx = call @swift_task_alloc(16)
call @coro.async.suspend(…)
call @swift_task_dealloc(%callee_ctxt)

%callee_ctx2 = call @swift_task_alloc(32)
call @coro.async.suspend(…)
call @swift_task_dealloc(%callee_ctxt2)
```

- Improve Coroutine frame entry use
  - Fix lifetime intrinsic based fragment local analysis (found bugs e.g https://reviews.llvm.org/D110953, https://reviews.llvm.org/D110949)
  - Live values: no sharing of slots for spills with disjoint liveness slots (only allocas)