Let's Go Coroutine

Daniel Hillerström (Huawei Zurich Research Center),
Luna Phipps-Costin (Northeastern University)

L(feat. WasmFX)e

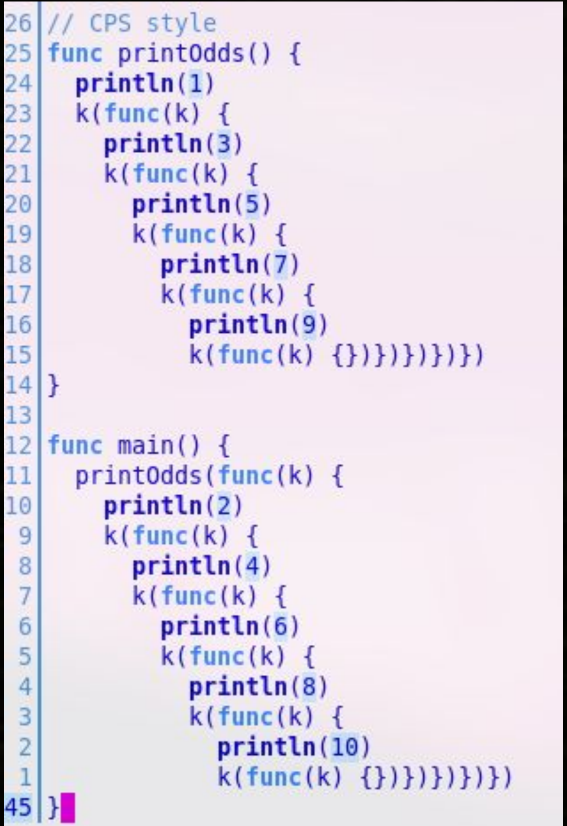
Sam Lindley, Andreas Rossberg, Daan Leijen, KC Sivaramakrishnan, Matija Pretnar, Arjun Guha

I have a Go program…

```
func printOdds() {
    println(1)
    println(3)
    println(5)
    println(7)
    println(9)
}

func main() {
    go printOdds()
    println(2)
    println(4)
    println(6)
    println(8)
    println(10)
}
```

- Coroutines central to the identity of the language
- Launched by **go** keyword
- (Cooperative concurrency)
- How do I compile this… ?

Use CPS of course!



```
// CPS style
func printOdds() {
  println(1)
  k(func(k) {
    println(3)
    k(func(k) {
      println(5)
      k(func(k) {
        println(7)
        k(func(k) {
          println(9)
          k(func(k) {})})})})})
}

func main() {
  printOdds(func(k) {
    println(2)
    k(func(k) {
      println(4)
      k(func(k) {
        println(6)
        k(func(k) {
          println(8)
          k(func(k) {
            println(10)
            k(func(k) {})})})})})})
}
```

(If you ask a compilers person)

Look familiar? (JS Promise)

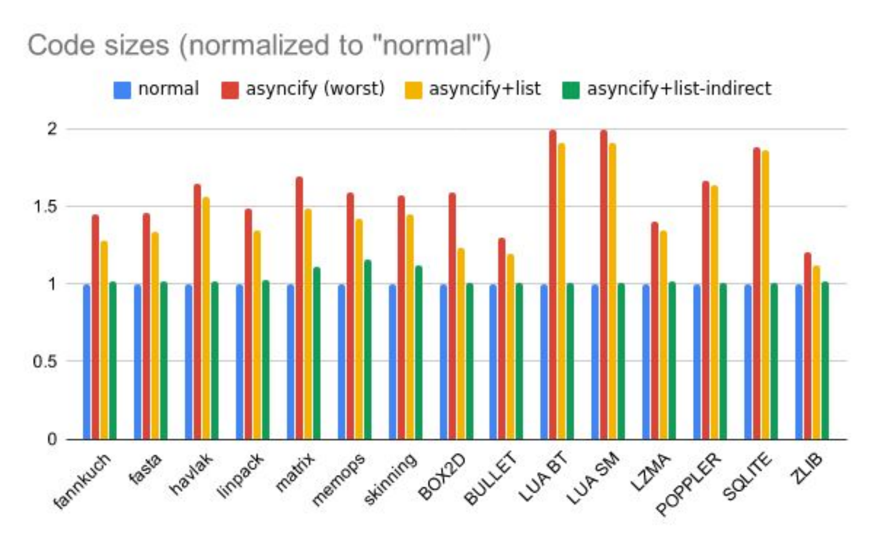
Hillerström et al 2017

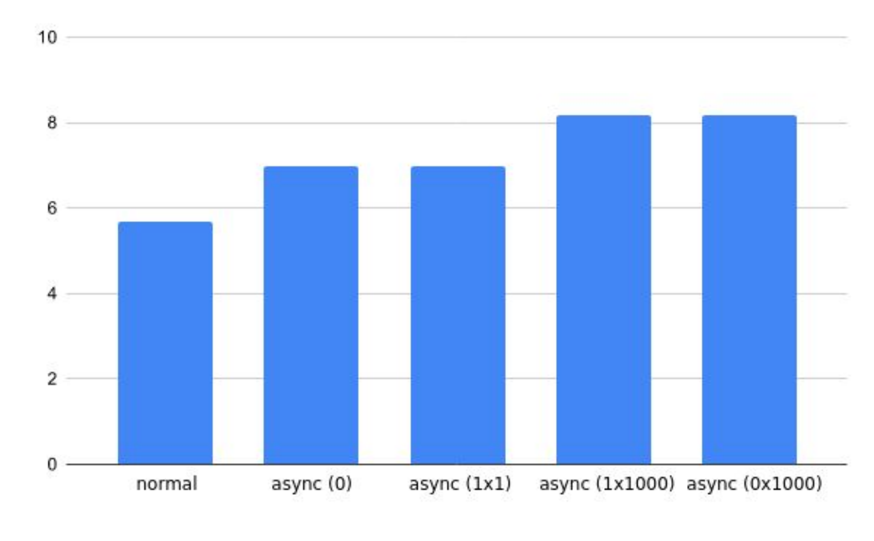
Or asyncify

- Alon Zakai / Emscripten people
- CPS at the WebAssembly level
    - Without lambda! Requires spilling locals to stack, branching on every call, etc…
- Some cost: time, program size, etc…
- Instrumentation is… complicated. For compiler, for asyncify, and for embedder

Or asyncify

- It works!





Runtime (lower is better)

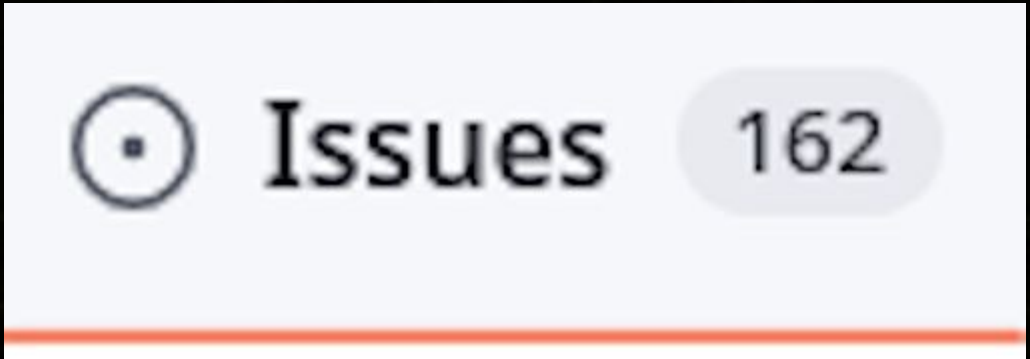
https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html

Or get your hands dirty

Obviously we can “just” extend Wasm with coroutines

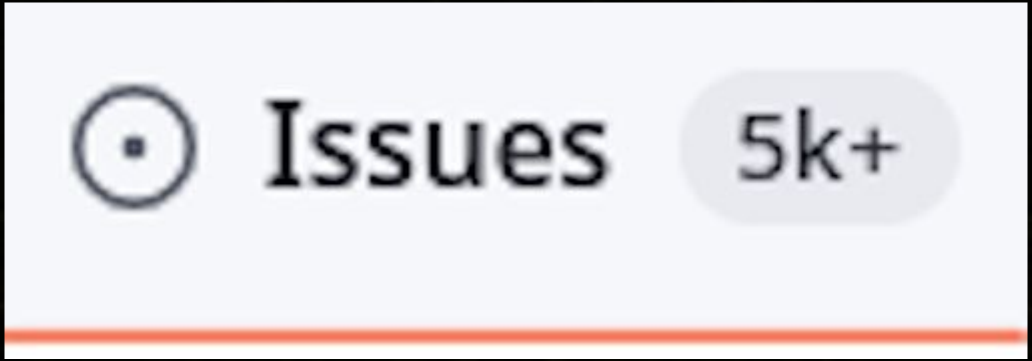
Or get your hands dirty

Obviously we can “just” extend Wasm with coroutines



Or get your hands dirty

Obviously we can “just” extend Wasm with coroutines



Or you can use WasmFX

DEMO

WasmFX at glance

- Extends Wasm with first-class control
- Delimited continuations controlled via effect handlers
- Minimal extension to Wasm (6 instructions + 1 type)
- Depends on function references and exception handling proposals
- Grounded in real world experience and research

https://wasmfx.dev

WasmFX deep dive: Continuation type

(cont $ft)

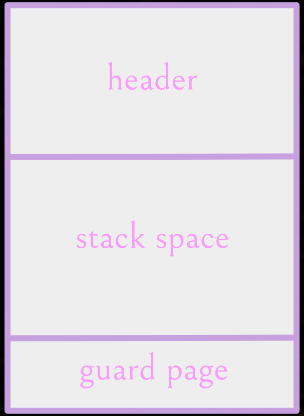
Reference type parameterised by a function type $ft : [s*] -> [t*]

WasmFX deep dive: Continuation allocation

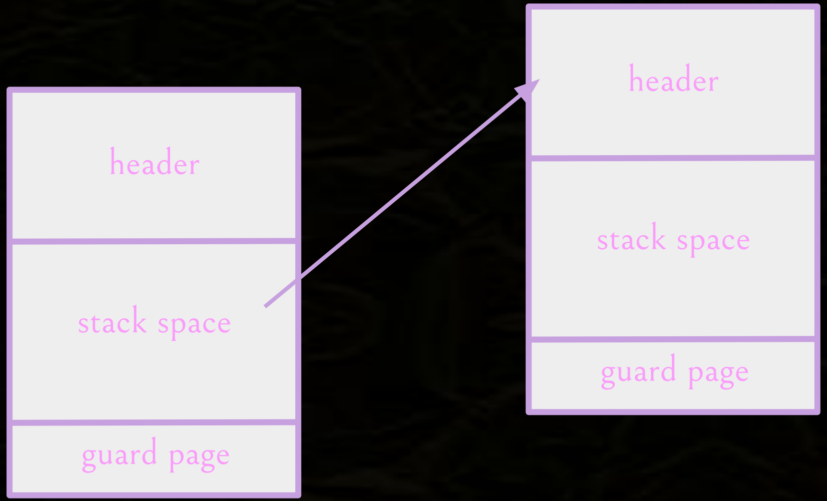
cont.new : [(ref null $ft)] -> [(ref $ct)]

where $ft : [s*] -> [t*] and $ct : cont $ft

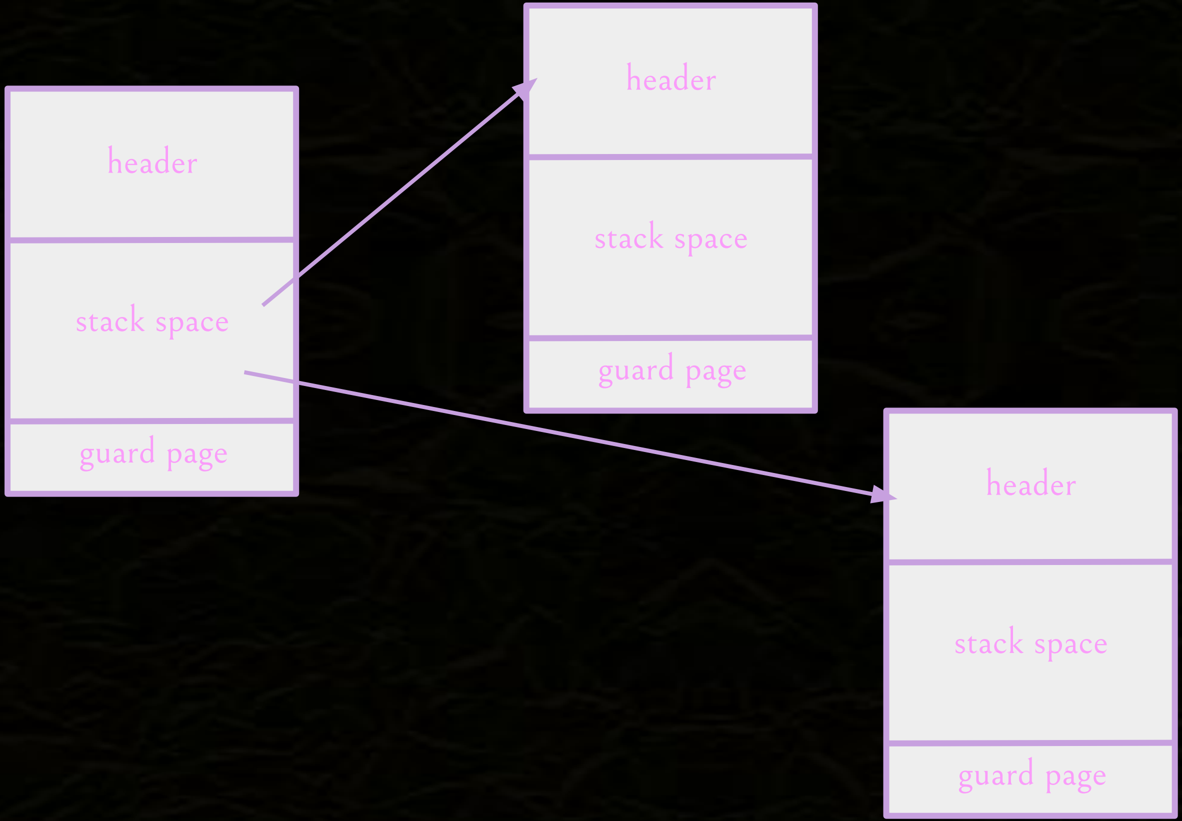
Thinking in terms of stacks: cont.new



Thinking in terms of stacks: cont.new



Thinking in terms of stacks: cont.new



WasmFX deep dive: Resumption

$$\text{resume (tag \$t \$h)}^* : [s^* \text{ (ref null \$ct)}] \text{ -> } [t^*]$$

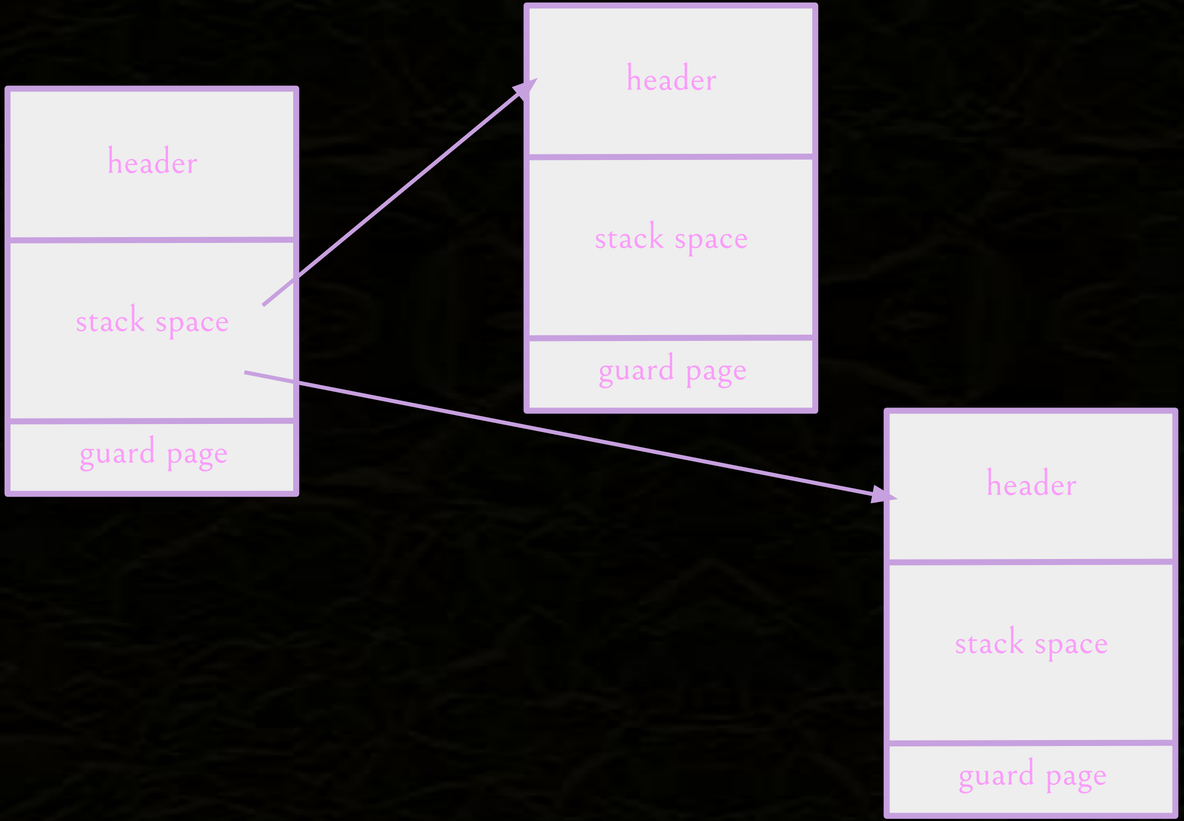
$\text{where } \{\$t_i : [s_i{}^*] \text{ -> } [t_i{}^*] \text{ and } \$h : [s_i{}^* \text{ (ref null } \$ct_i)] \text{ and}$

$\qquad \$ct_i : \text{cont } \$ft_i \text{ and } \$ft_i : [t_i{}^*] \text{ -> } [t^*]\ \}_i$

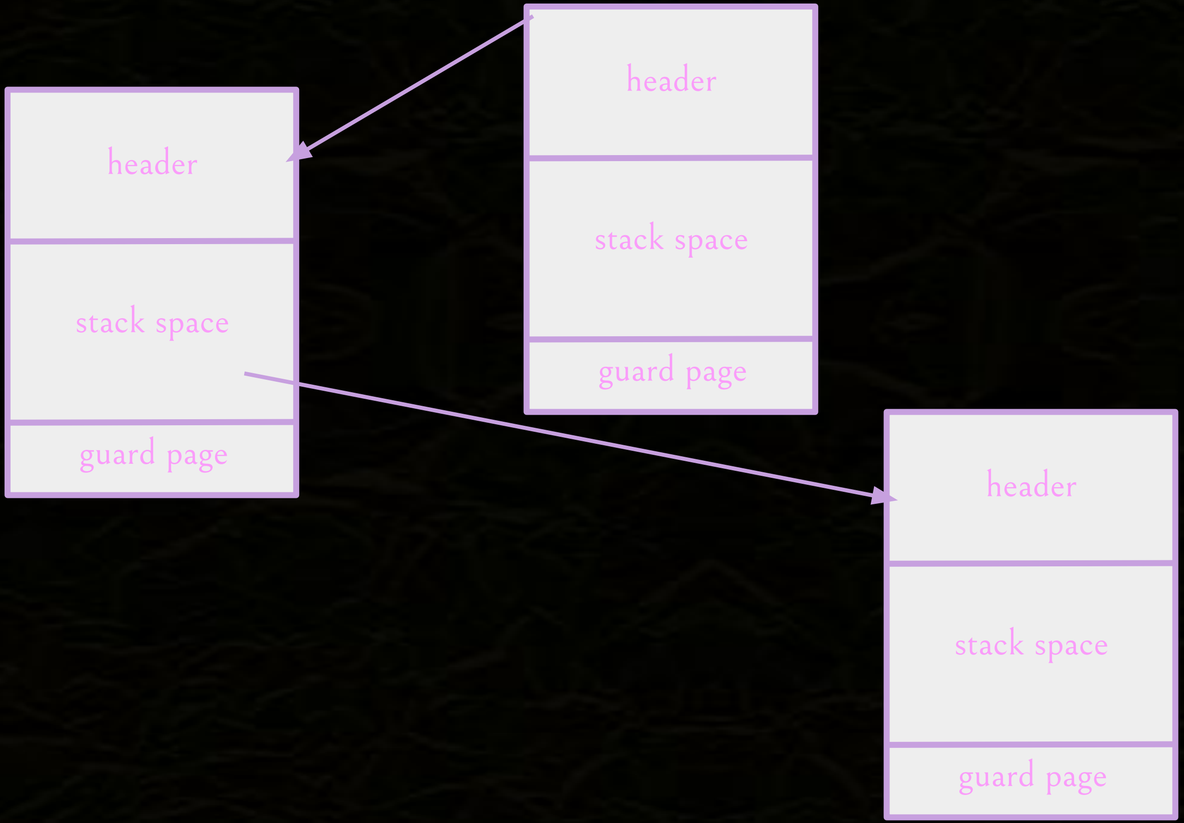
$\text{and } \$ct : \text{cont } \$ft$

$\quad \$ft : [s^*] \text{ -> } [t^*]$

Thinking in terms of stacks: resume



Thinking in terms of stacks: resume

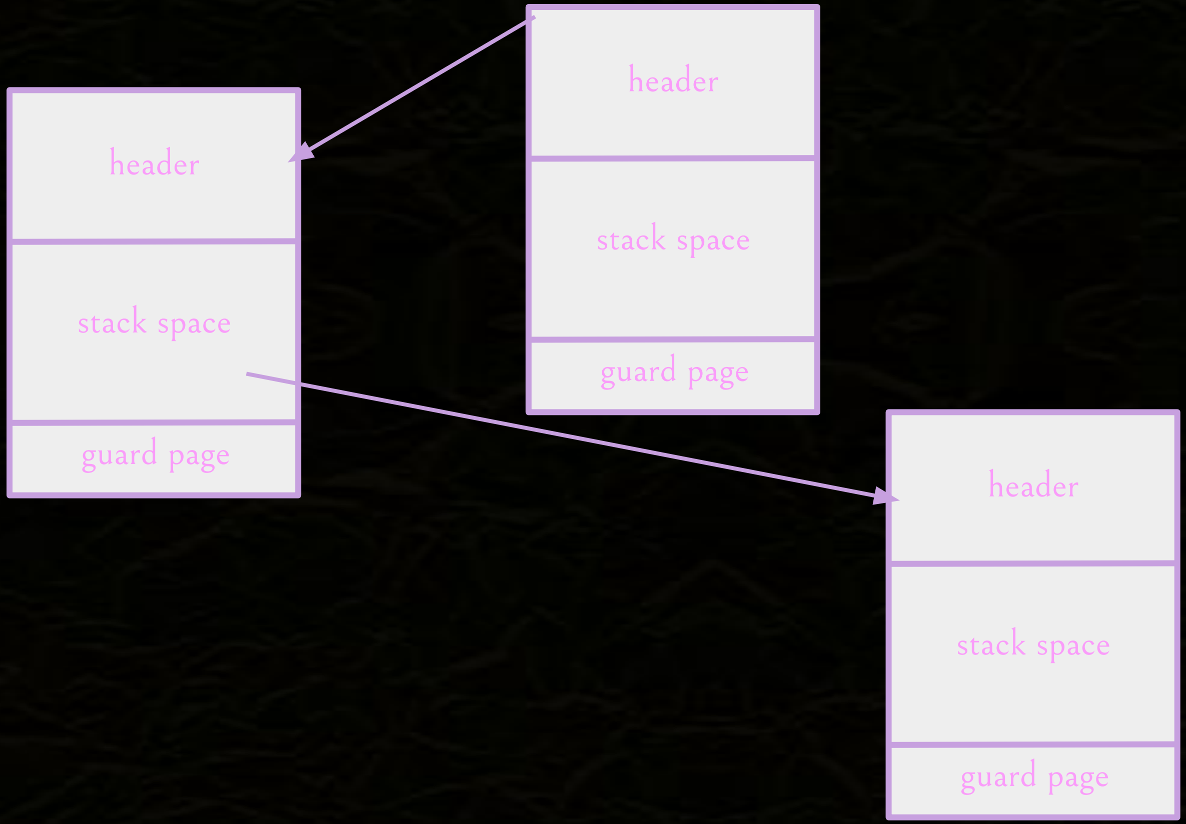


WasmFX deep dive: Suspension

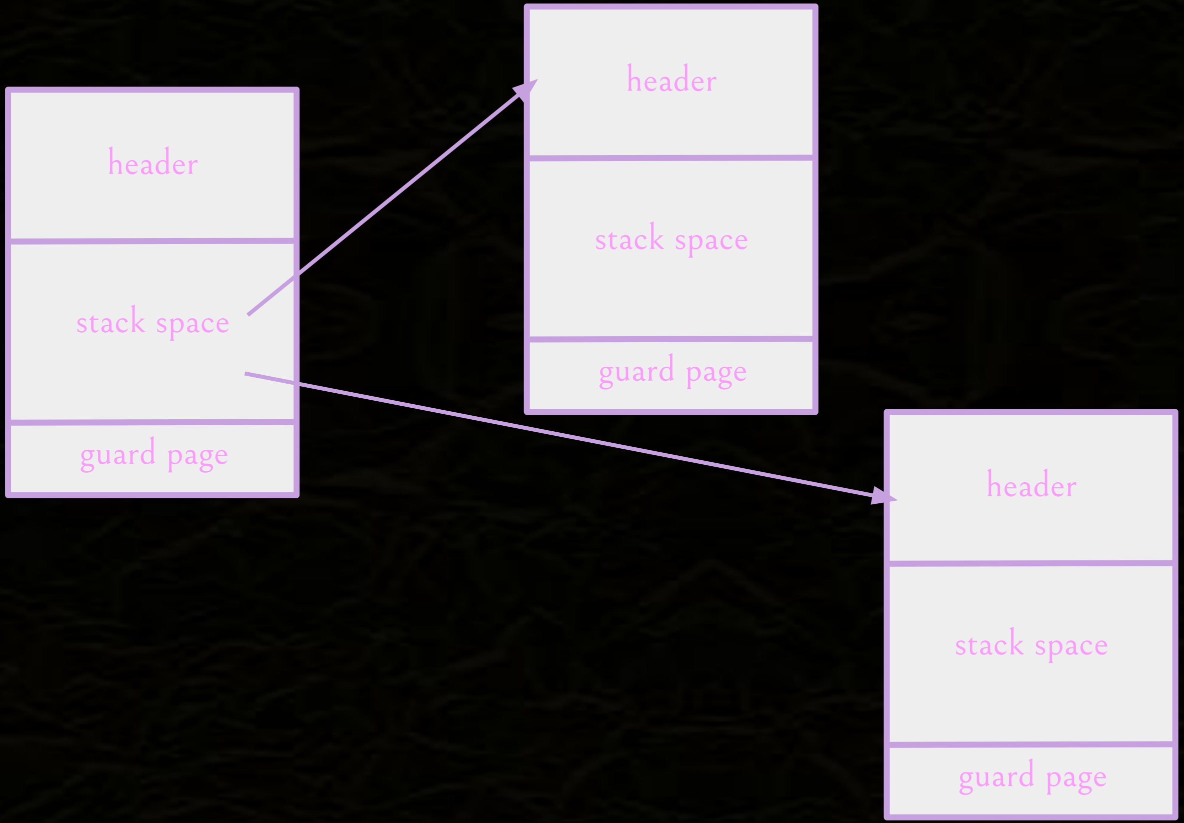
suspend $tag : [s*] -> [t*]

where $tag : [s*] -> [t*]

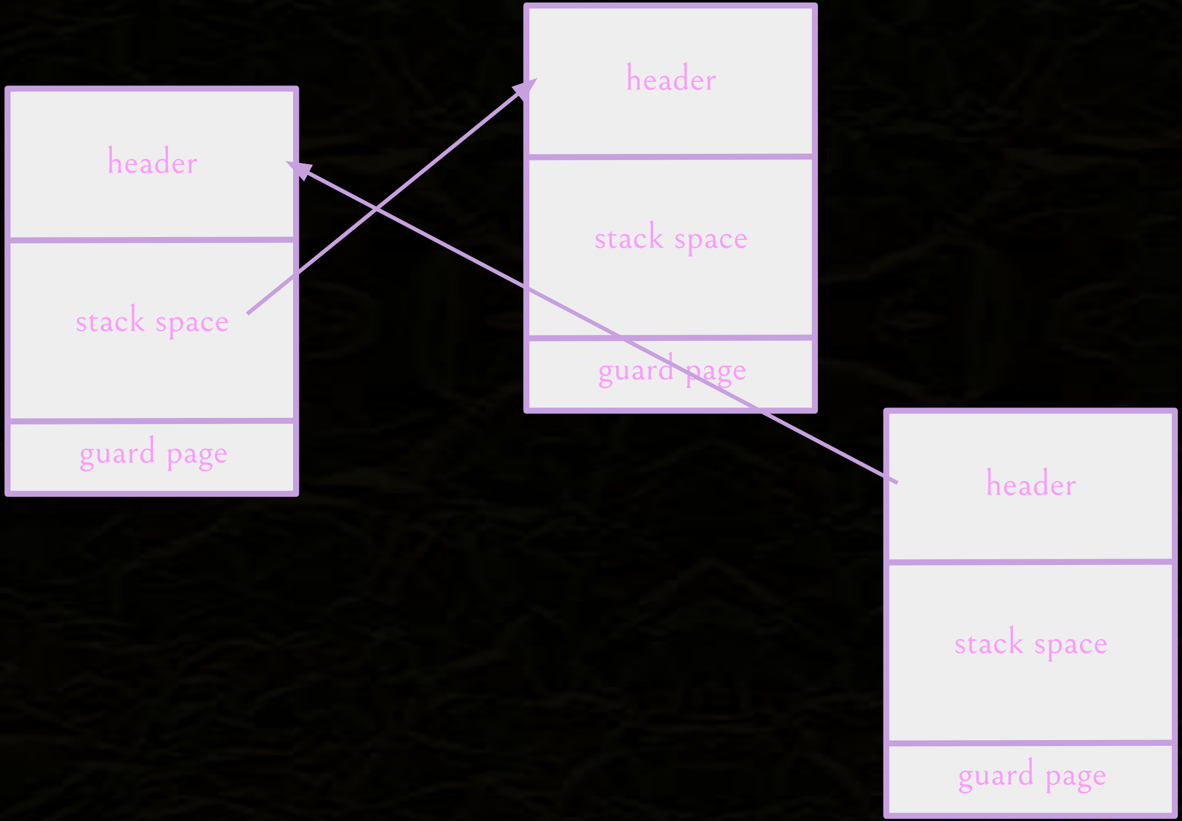
Thinking in terms of stacks: suspend



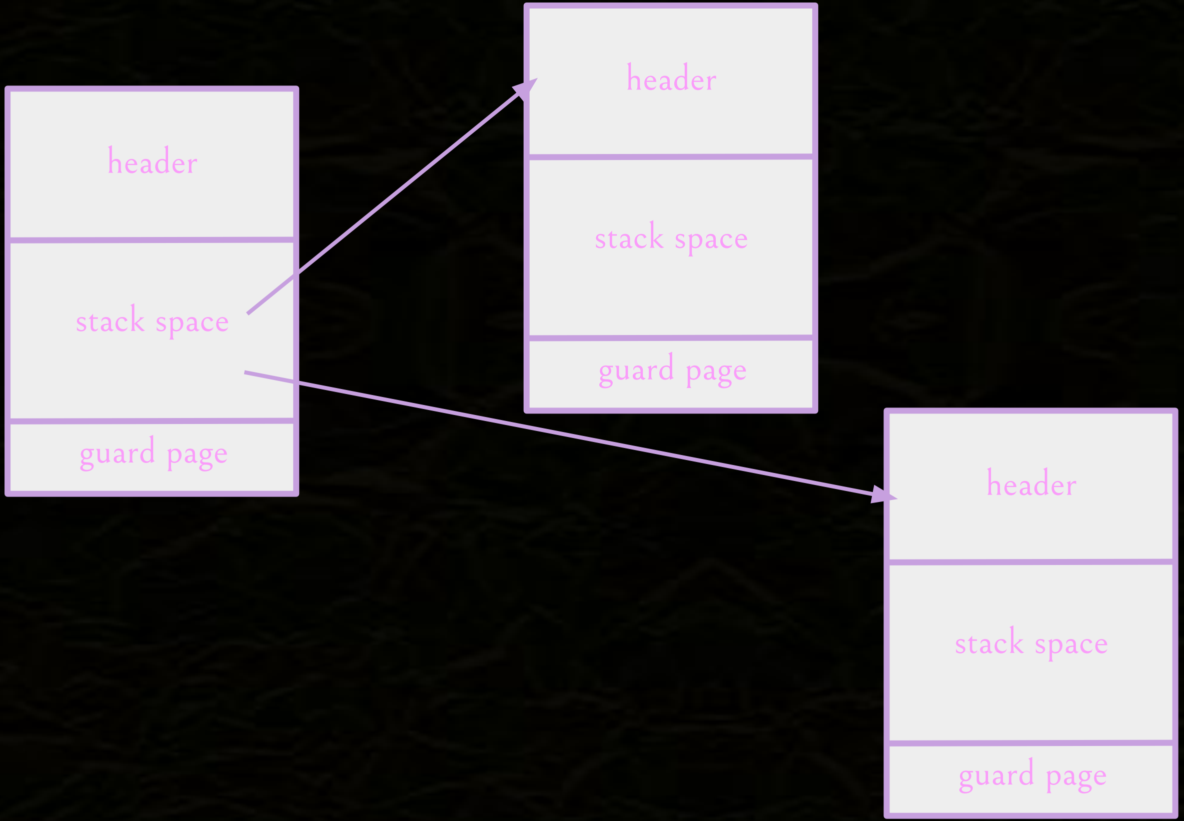
Thinking in terms of stacks: suspend



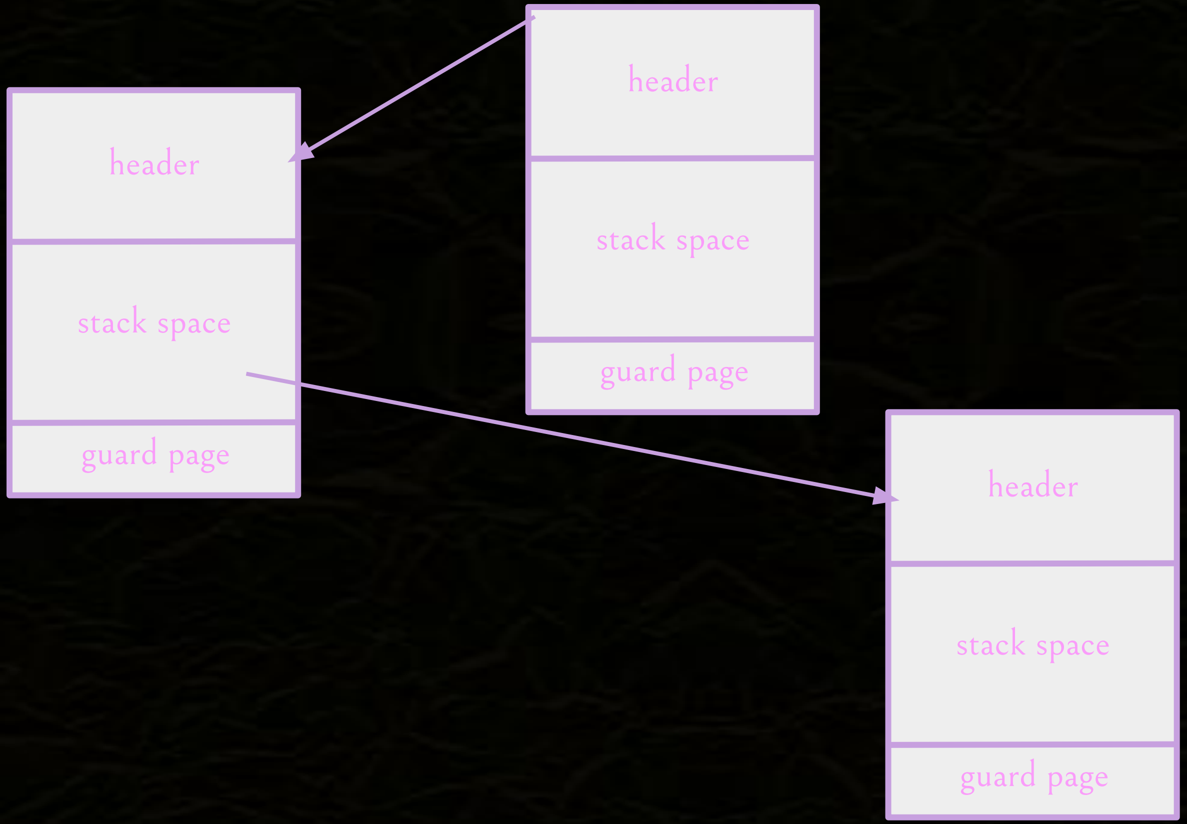
Thinking in terms of stacks: suspend



Thinking in terms of stacks: suspend



Thinking in terms of stacks: suspend



WasmFX deep dive: cont.bind & barrier

```
cont.bind (type $ct) : [s* (ref null $ct)] -> [(ref $ct’)]

resume_throw (tag $exn) (tag $t $h)* : [s* (ref null $ct)] -> [t*]

barrier $l (type $bt) instr* : [s*] -> [t*]
```

Let's return to our program...

```
func printOdds() {
    println(1)
    println(3)
    println(5)
    println(7)
    println(9)
}

func main() {
    go printOdds()
    println(2)
    println(4)
    println(6)
    println(8)
    println(10)
}
```

- Let's imagine we wrote this in WasmFX...

Let's return to our program...

```
(tag $scheduler)

(type $crt (cont $unit_unit))

(type $newcrt (cont $i32_i32_unit))

(table $queue) (func $enqueue ...) (func $dequeue ...)
```

Let’s return to our program...

```
(func $runtime.scheduler
  block $done
    loop $mainloop
      ... ;; (check schedulerDone and exit)
      call $dequeue
      br_on_null $done
      (block $coroutine_suspend (param (ref $crt)) (result (ref $crt))
        (resume
          (tag $scheduler $coroutine_suspend))
        br $mainloop)
      call $enqueue
      br $mainloop
    end
  end)
```

Let’s return to our program...

```
(func $internal/task.Pause
  suspend $scheduler)


(func $lift_call_indirect (param i32 i32) ...)
```

Let's return to our program…

```
(func $internal/task.start (param $fn i32) (param $args i32)
  local.get $fn
  local.get $args
  (cont.new (type $newcrt) (ref.func $lift_call_indirect))
  (cont.bind (type $crt))
  call $enqueue)
```

It turns out we can do exactly that!

I’ve just showed you the entire runtime we drop in to tinygo!

A compiler from Go to WasmFX

- TinyGo: Go subset, clean slate implementation using LLVM
- 11 line change to the compiler:
    - Don't run asyncify (don't need it!)
    - Insert a placeholder in the middle of a runtime function
- In defense of writing a compiler in Perl:
    - Only two parsers at this time support WasmFX
    - Just replace Tinygo's runtime with our own, in WasmFX text format!
    - *WasmFX makes this really easy*

WasmFX in WasmTime

“A fast and secure runtime for WebAssembly” - in particular, non-browser-based

- “optimizing Cranelift code generator”
- WASI + standards compliant

WasmFX implementation in Wasmtime

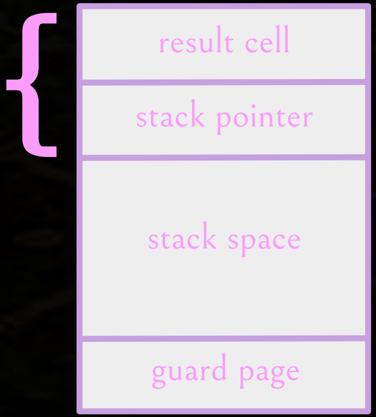
- Wasmtime provides a fiber abstraction
- Holds a “stack” - a real system stack, matching a suspended computation
- High-level API:
  - new
  - suspend
  - resume
- Even if they didn’t exist, libmprompt does!
- 😈 Nothing but hand-written assembly will do the job!

Wasmtime fiber interface

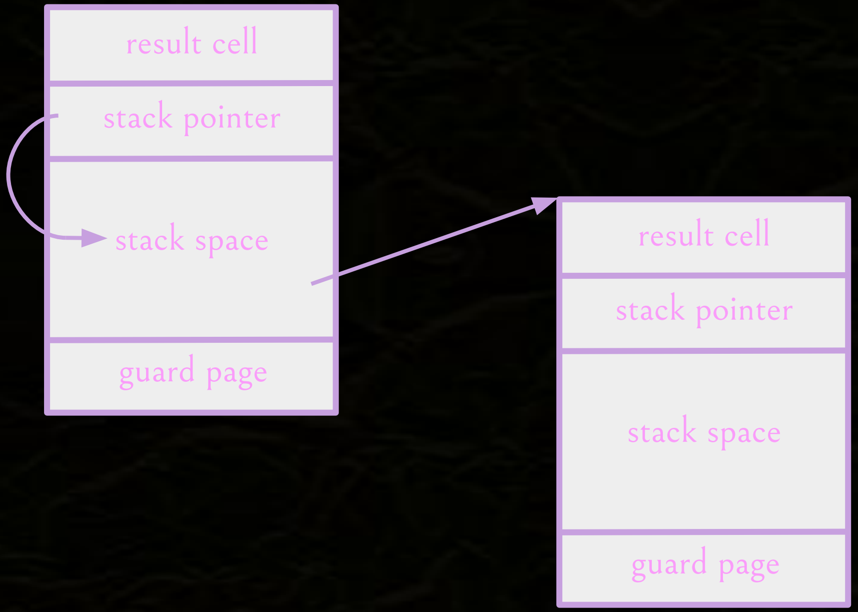
The essence of the Wasmtime fiber interface in Rust

```
trait FiberStack {
  fn new(size: usize) -> io::Result<Self>
}

trait<Resume, Yield, Return> Fiber<Resume, Yield, Return> {
  fn new(stack: FiberStack,
         func: FnOnce(Resume, &Suspend<Resume, Yield, Return>) -> Return
  fn resume(&self, val: Resume) -> Result<Return, Yield>
}

trait Suspend<Resume, Yield, Return> {
  fn suspend(&self, Yield) -> Resume
}
```

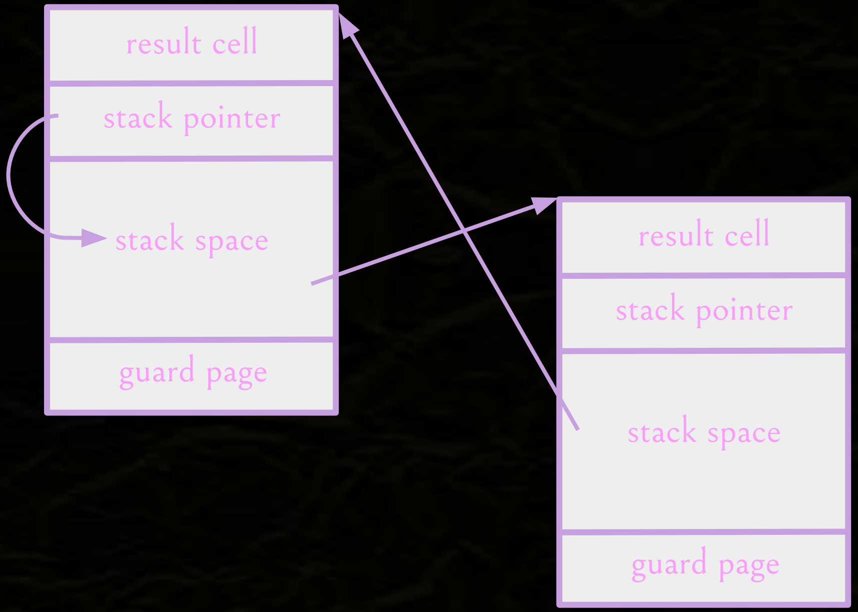
Wasmtime Fibers: Stack layout



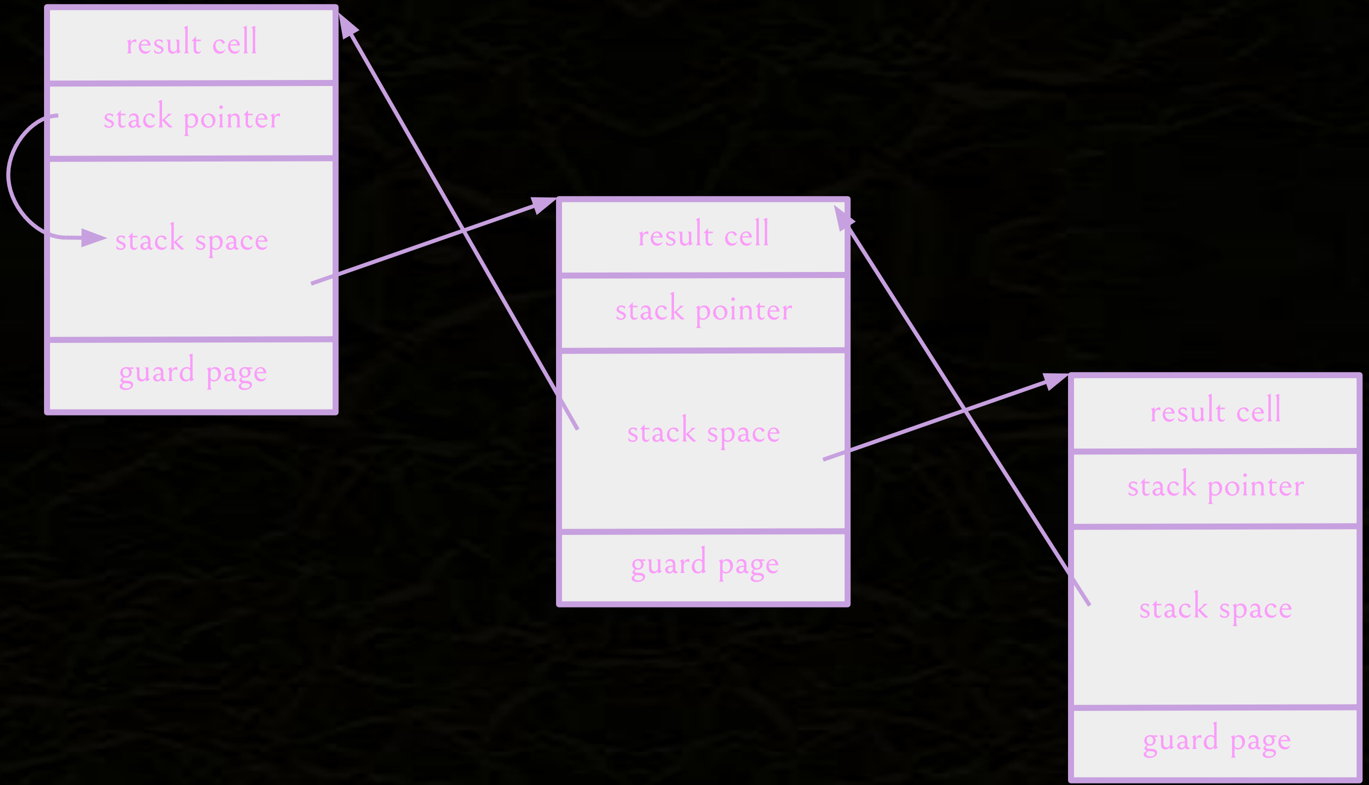
Wasmtime Fibers: Create fiber



Wasmtime Fibers: Resume & suspend fiber



Wasmtime Fibers: Nesting fibers



Wasmtime Fibers

```
.wasmtime_fiber_switch:
      // Save callee-saved registers
      push …

      // Load resume pointer from header, save previous
      mov rax, -0x20[rdi]
      mov -0x20[rdi], rsp

      // Swap stacks and restore callee-saved registers
      mov rsp, rax
      pop …
       ret
```

cont.suspend

- We keep a reference to the current stack in the context
- Maintain a stack’s parent at the top of the stack

cont.resume

Fibers provide one handler to suspend to, but we need to find our tag!

- Suspend provides the tag index, we desugar to br_table
- Completion gives a special sentinel value
- Plan: on default, we suspend to our parent with the same values

Passing values in and out of stacks not yet supported by Wasmtime

- Plan: box and pass a pointer. Various trampoline nonsense

The gist of encoding effect handlers on top of Wasmtime fibers

Fix suitably *Resume*, *Yield*, and *Return* types.

Continuation creation $\mathcal{I}[\![-]\!] : \mathsf{Instr} \times \mathsf{ValStack} \to \mathsf{Rust}$

$$\mathcal{I}[\![\texttt{\textbf{cont.new}}; [f]]\!] = \texttt{Fiber.new(FiberStack.new(STACK\_SIZE), |resume, \&mySuspend| \{Return(f(resume))\})}$$

Continuation resumption $\mathcal{T}[\![-]\!] : \mathsf{Tag} \to \mathsf{Rust}$, $\mathcal{L}[\![-]\!] : \mathsf{Label} \times \mathsf{ValStack} \to \mathsf{Rust}$

$$\begin{aligned} & \mathcal{I}[\![\textbf{resume}\ (\textbf{tag}\ \$tag\ \$h)^*; [x_0, \ldots, x_n, k]]\!] \\ = \ & \texttt{\textbf{match} Fiber.resume(k, Tuple(}x_0, \ldots, x_n\texttt{)) \{} \\ & \quad [\texttt{Yield(Op(}\mathcal{T}[\![\$tag_i]\!]\texttt{, args)) => }\mathcal{L}[\![\$h_i; [args, k]]\!]]_i \\ & \quad \texttt{Yield(Op(tag, args)) => Fiber.resume(k, mySuspend.suspend(Op(tag, args)))} \\ & \quad \texttt{Return(x) => x} \\ & \texttt{\}} \end{aligned}$$

Continuation suspension

$$\mathcal{I}[\![\textbf{suspend}; [tag, args]]\!] = \texttt{mySuspend.suspend(Op(tag,args))}$$

Let's Go Coroutine

- WasmFX: Effect handlers for wasm!
- We can compile it!
- We can produce it!
- Let’s go coroutine!

github.com/effect-handlers

wasmfx.dev

Next up: benchmarking!

THANK YOU!