Compiling WasmFX: *Is It Hard?*

Daniel Hillerström (The University of Edinburgh),
Luna Phipps-Costin (Northeastern University)

WebAssembly + Typed Continuations + Wasmtime

“WebAssembly is a binary instruction format for a stack-based virtual machine.”

“The WasmFX project extends WebAssembly with effect handlers as a unifying mechanism to enable efficient compilation of control idioms, such as async/await, generators/iterators, first-class continuations, etc.”

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

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

What do we need?

Dependencies:

- Function references (dependency)
- Exceptions (dependency)

Instructions:

- cont.new
- suspend
- resume
- cont.bind
- resume_throw
- barrier

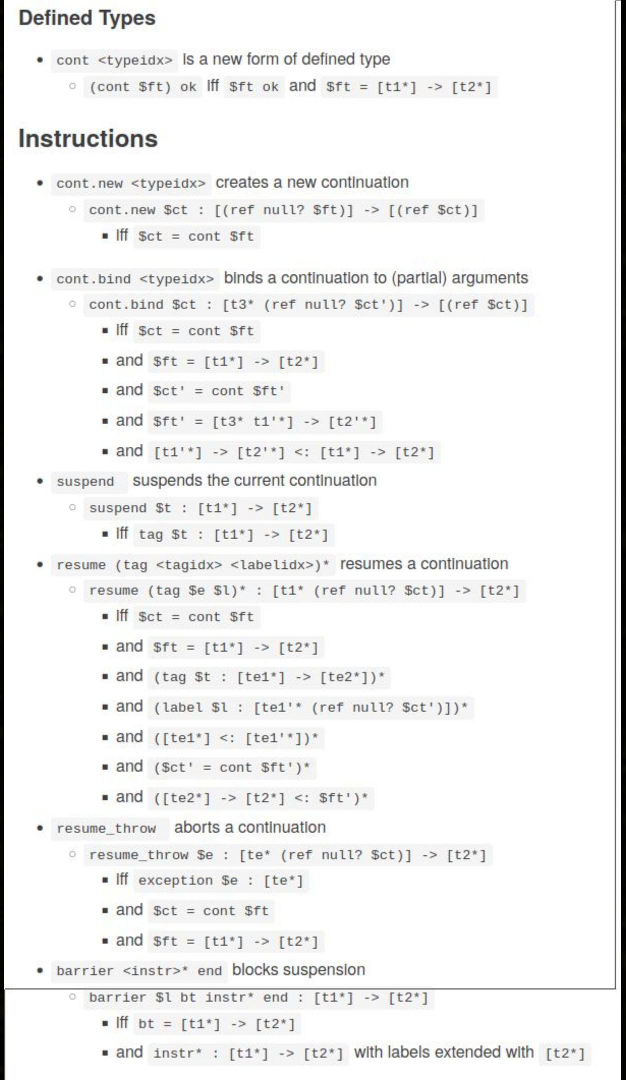
What do we need? Ask the spec

Dependencies:

- Function references (dependency)
- Exceptions (dependency)

Instructions:

- cont.new
- suspend
- resume
- cont.bind
- resume_throw
- barrier



Defined Types

- `cont <typeidx>` is a new form of defined type
  - `(cont $ft) ok` iff `$ft ok` and `$ft = [t1*] -> [t2*]`

Instructions

- `cont.new <typeidx>` creates a new continuation
  - `cont.new $ct : [(ref null? $ft)] -> [(ref $ct)]`
    - iff `$ct = cont $ft`
- `cont.bind <typeidx>` binds a continuation to (partial) arguments
  - `cont.bind $ct : [t3* (ref null? $ct')] -> [(ref $ct)]`
    - iff `$ct = cont $ft`
    - and `$ft = [t1*] -> [t2*]`
    - and `$ct' = cont $ft'`
    - and `$ft' = [t3* t1'*] -> [t2'*]`
    - and `[t1'*] -> [t2'*] <: [t1*] -> [t2*]`
- `suspend` suspends the current continuation
  - `suspend $t : [t1*] -> [t2*]`
    - iff `tag $t : [t1*] -> [t2*]`
- `resume (tag <tagidx> <labelidx>)*` resumes a continuation
  - `resume (tag $e $l)* : [t1* (ref null? $ct)] -> [t2*]`
    - iff `$ct = cont $ft`
    - and `$ft = [t1*] -> [t2*]`
    - and `(tag $t : [te1*] -> [te2*])*`
    - and `(label $l : [te1'* (ref null? $ct')])*`
    - and `([te1*] <: [te1'*])*`
    - and `($ct' = cont $ft')*`
    - and `([te2*] -> [t2*] <: $ft')*`
- `resume_throw` aborts a continuation
  - `resume_throw $e : [te* (ref null? $ct)] -> [t2*]`
    - iff `exception $e : [te*]`
    - and `$ct = cont $ft`
    - and `$ft = [t1*] -> [t2*]`
- `barrier <instr>* end` blocks suspension
  - `barrier $l bt instr* end : [t1*] -> [t2*]`
    - iff `bt = [t1*] -> [t2*]`
    - and `instr* : [t1*] -> [t2*]` with labels extended with `[t2*]`

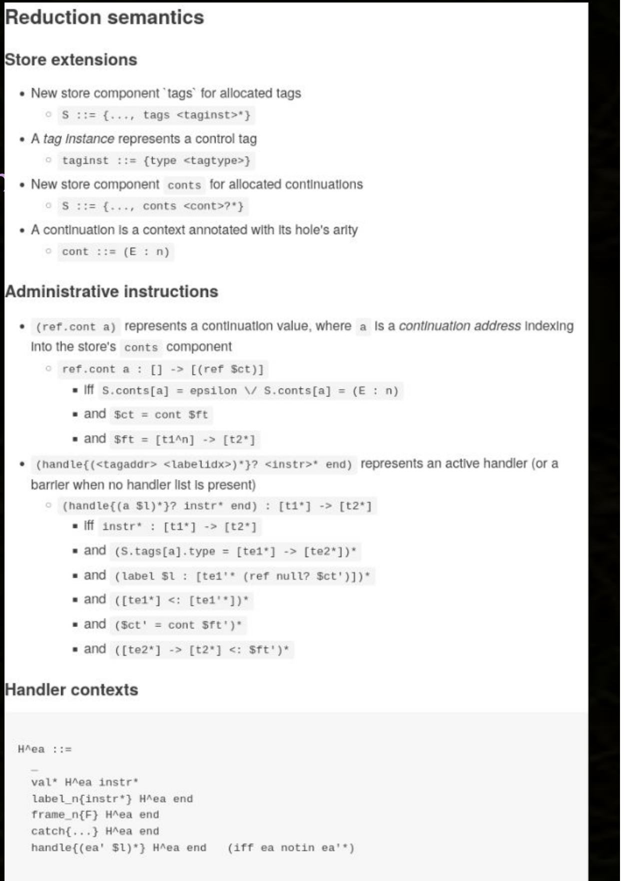
What do we need? Ask the spec?

Dependencies:

- Function references (depen
- Exceptions (dependency)

Instructions:

- cont.new
- suspend
- resume
- cont.bind
- resume_throw
- barrier



Reduction semantics

Store extensions

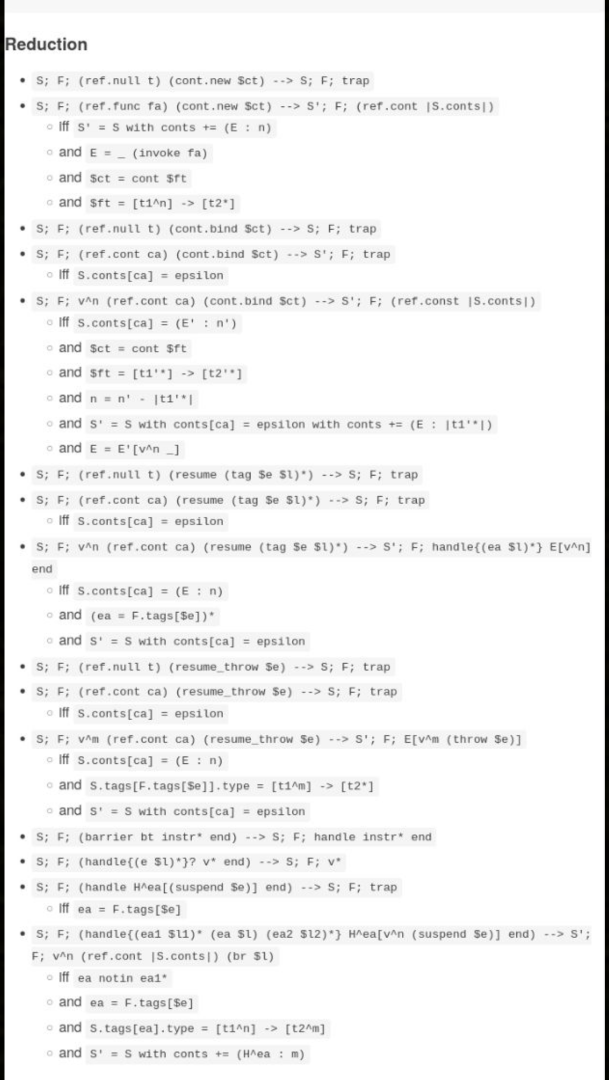
- New store component `tags` for allocated tags
  - `S ::= {..., tags <taginst>*}`
- A *tag instance* represents a control tag
  - `taginst ::= {type <tagtype>}`
- New store component `conts` for allocated continuations
  - `S ::= {..., conts <cont>?*}`
- A continuation is a context annotated with its hole's arity
  - `cont ::= (E : n)`

Administrative instructions

- `(ref.cont a)` represents a continuation value, where `a` is a *continuation address* indexing into the store's `conts` component
  - `ref.cont a : [] -> [(ref $ct)]`
    - iff `S.conts[a] = epsilon \/ S.conts[a] = (E : n)`
    - and `$ct = cont $ft`
    - and `$ft = [t1^n] -> [t2*]`
- `(handle{(<tagaddr> <labelidx>)*}? <instr>* end)` represents an active handler (or a barrier when no handler list is present)
  - `(handle{(a $l)*}? instr* end) : [t1*] -> [t2*]`
    - iff `instr* : [t1*] -> [t2*]`
    - and `(S.tags[a].type = [te1*] -> [te2*])*`
    - and `(label $l : [te1'* (ref null? $ct')])*`
    - and `([te1*] <: [te1'*])*`
    - and `($ct' = cont $ft')*`
    - and `([te2*] -> [t2*] <: $ft')*`

Handler contexts

```
H^ea ::=
  _
  val* H^ea instr*
  label_n{instr*} H^ea end
  frame_n{F} H^ea end
  catch{...} H^ea end
  handle{(ea' $l)*} H^ea end   (iff ea notin ea'*)
```



Reduction

- `S; F; (ref.null t) (cont.new $ct) --> S; F; trap`
- `S; F; (ref.func fa) (cont.new $ct) --> S'; F; (ref.cont |S.conts|)`
  - iff `S' = S with conts += (E : n)`
  - and `E = _ (invoke fa)`
  - and `$ct = cont $ft`
  - and `$ft = [t1^n] -> [t2*]`
- `S; F; (ref.null t) (cont.bind $ct) --> S; F; trap`
- `S; F; (ref.cont ca) (cont.bind $ct) --> S'; F; trap`
  - iff `S.conts[ca] = epsilon`
- `S; F; v^n (ref.cont ca) (cont.bind $ct) --> S'; F; (ref.const |S.conts|)`
  - iff `S.conts[ca] = (E' : n')`
  - and `$ct = cont $ft`
  - and `$ft = [t1'*] -> [t2'*]`
  - and `n = n' - |t1'*|`
  - and `S' = S with conts[ca] = epsilon with conts += (E : |t1'*|)`
  - and `E = E'[v^n _]`
- `S; F; (ref.null t) (resume (tag $e $l)*) --> S; F; trap`
- `S; F; (ref.cont ca) (resume (tag $e $l)*) --> S; F; trap`
  - iff `S.conts[ca] = epsilon`
- `S; F; v^n (ref.cont ca) (resume (tag $e $l)*) --> S'; F; handle{(ea $l)*} E[v^n] end`
  - iff `S.conts[ca] = (E : n)`
  - and `(ea = F.tags[$e])*`
  - and `S' = S with conts[ca] = epsilon`
- `S; F; (ref.null t) (resume_throw $e) --> S; F; trap`
- `S; F; (ref.cont ca) (resume_throw $e) --> S; F; trap`
  - iff `S.conts[ca] = epsilon`
- `S; F; v^m (ref.cont ca) (resume_throw $e) --> S'; F; E[v^m (throw $e)]`
  - iff `S.conts[ca] = (E : n)`
  - and `S.tags[F.tags[$e]].type = [t1^m] -> [t2*]`
  - and `S' = S with conts[ca] = epsilon`
- `S; F; (barrier bt instr* end) --> S; F; handle instr* end`
- `S; F; (handle{(e $l)*}? v* end) --> S; F; v*`
- `S; F; (handle H^ea[(suspend $e)] end) --> S; F; trap`
  - iff `ea = F.tags[$e]`
- `S; F; (handle{(ea1 $l1)* (ea $l) (ea2 $l2)*} H^ea[v^n (suspend $e)] end) --> S'; F; v^n (ref.cont |S.conts|) (br $l)`
  - iff `ea notin ea1*`
  - and `ea = F.tags[$e]`
  - and `S.tags[ea].type = [t1^n] -> [t2^m]`
  - and `S' = S with conts += (H^ea : m)`

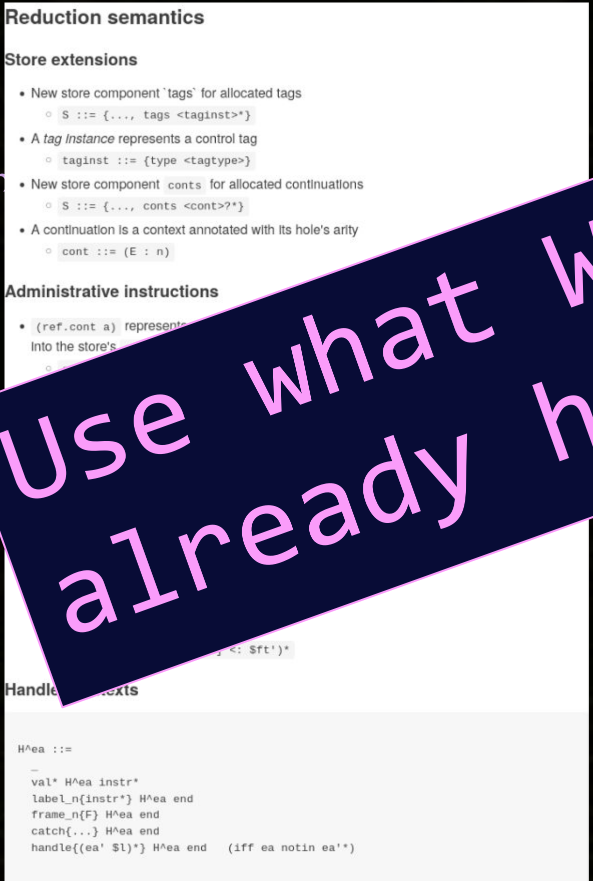
What do we need? Ask the spec?

Dependencies:

- Function references (depen
- Exceptions (dependency)

Instructions:

- cont.new
- suspend
- resume
- cont.bind
- resume_throw
- barrier



Reduction semantics

Store extensions

- New store component `tags` for allocated tags
  - `S ::= {..., tags <taginst>*}`
- A *tag instance* represents a control tag
  - `taginst ::= {type <tagtype>}`
- New store component `conts` for allocated continuations
  - `S ::= {..., conts <cont>?*}`
- A continuation is a context annotated with its hole's arity
  - `cont ::= (E : n)`

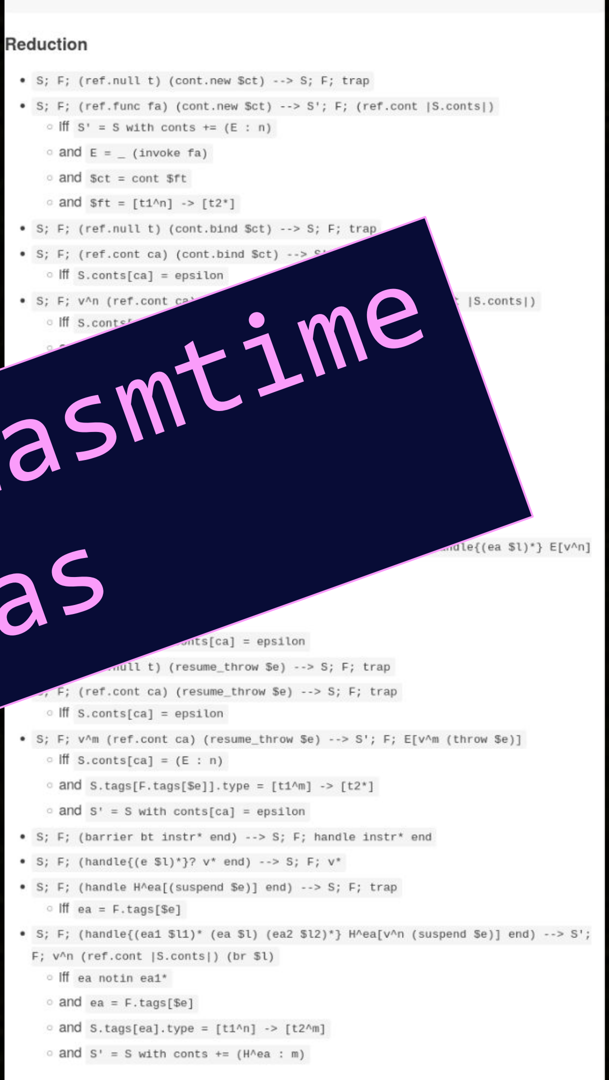
Administrative instructions

- `(ref.cont a)` represent
  into the store's

`<: $ft')*`

Handle xts

```
H^ea ::=
  _
  val* H^ea instr*
  label_n{instr*} H^ea end
  frame_n{F} H^ea end
  catch{...} H^ea end
  handle{(ea' $l)*} H^ea end   (iff ea notin ea'*)
```



Reduction

- `S; F; (ref.null t) (cont.new $ct) --> S; F; trap`
- `S; F; (ref.func fa) (cont.new $ct) --> S'; F; (ref.cont |S.conts|)`
  - Iff `S' = S with conts += (E : n)`
  - and `E = _ (invoke fa)`
  - and `$ct = cont $ft`
  - and `$ft = [t1^n] -> [t2*]`
- `S; F; (ref.null t) (cont.bind $ct) --> S; F; trap`
- `S; F; (ref.cont ca) (cont.bind $ct) --> S`
  - Iff `S.conts[ca] = epsilon`
- `S; F; v^n (ref.cont ca` `|S.conts|)`
  - Iff `S.conts`

`ndle{(ea $l)*} E[v^n]`

`nts[ca] = epsilon`

`null t) (resume_throw $e) --> S; F; trap`

- `F; (ref.cont ca) (resume_throw $e) --> S; F; trap`
  - Iff `S.conts[ca] = epsilon`
- `S; F; v^m (ref.cont ca) (resume_throw $e) --> S'; F; E[v^m (throw $e)]`
  - Iff `S.conts[ca] = (E : n)`
  - and `S.tags[F.tags[$e]].type = [t1^m] -> [t2*]`
  - and `S' = S with conts[ca] = epsilon`
- `S; F; (barrier bt instr* end) --> S; F; handle instr* end`
- `S; F; (handle{(e $l)*}? v* end) --> S; F; v*`
- `S; F; (handle H^ea[(suspend $e)] end) --> S; F; trap`
  - Iff `ea = F.tags[$e]`
- `S; F; (handle{(ea1 $l1)* (ea $l) (ea2 $l2)*} H^ea[v^n (suspend $e)] end) --> S'; F; v^n (ref.cont |S.conts|) (br $l)`
  - Iff `ea notin ea1*`
  - and `ea = F.tags[$e]`
  - and `S.tags[ea].type = [t1^n] -> [t2^m]`
  - and `S' = S with conts += (H^ea : m)`

Use what Wasmtime already has

What do we *really* need?

Dependencies:

- Function references (dependency) - text parsing exists!
- ~~Exceptions (dependency)~~

Instructions:

- cont.new - allocate a fiber
- suspend - just suspend!
- resume - resume a fiber, handle suspensions
- cont.bind
- resume_throw
- barrier

Demo

Function References - *is it hard?*

Yes!

- The type syntax changes require a complete refactor of wasmtime
- Value types themselves are now dependent on the context
- Subtyping
- Specification changes:
  - let / func.bind
  - call_ref annotation
- Specification / interpreter bugs / typos:
  - Table text syntax
- Working on upstreaming

Typing Typed Continuations - *is it hard?*

No!

- Bulk of the foundational changes covered by function references
- Spec has matched implementation / tests / reason

Adding a continuation value - *is it hard?*

Yes!

- Continuations and typed function references have no syntactic differentiation
- As a result, we need a context even to know what sort of value we have
- This requires a change of assumptions in calling in and out of Wasmtime

Typed continuations - *is it hard?*

Not as hard as it could be…

Because…

Wasmtime Fibers

- (Flavor of) limited multi-prompt delimited continuations
- 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!
- 😈 All of those libcalls will be slow! Nothing but hand-written assembly will do the job!
  - I/O bound in important contexts, inlining, compare table_grow etc., or: thanks, maybe later!

Wasmtime fiber interface

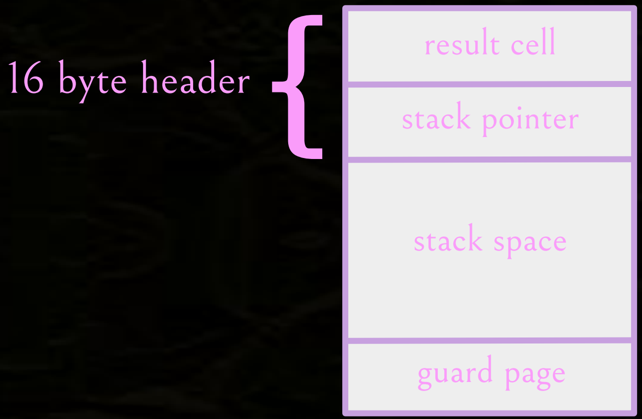
The essence of the Wasmtime fiber interface in Rust

```
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



16 byte header

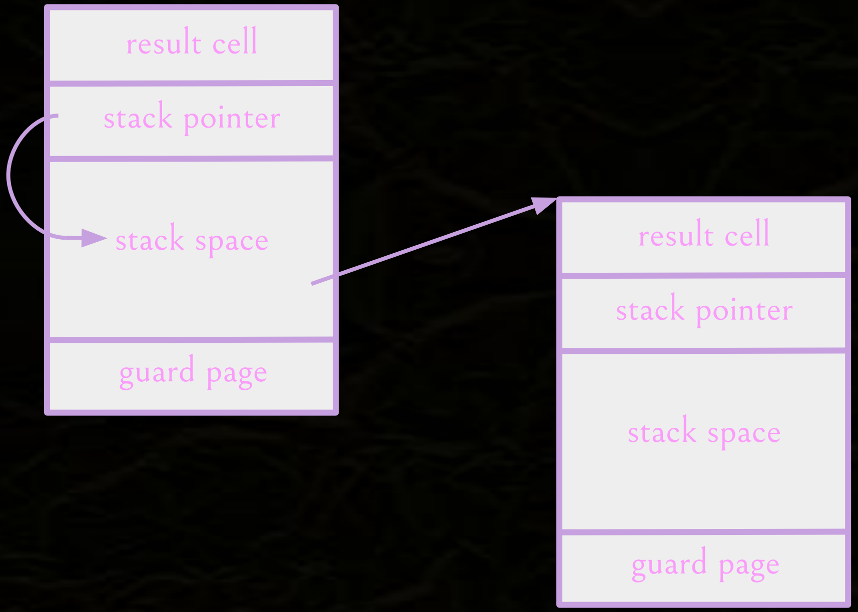
result cell

stack pointer

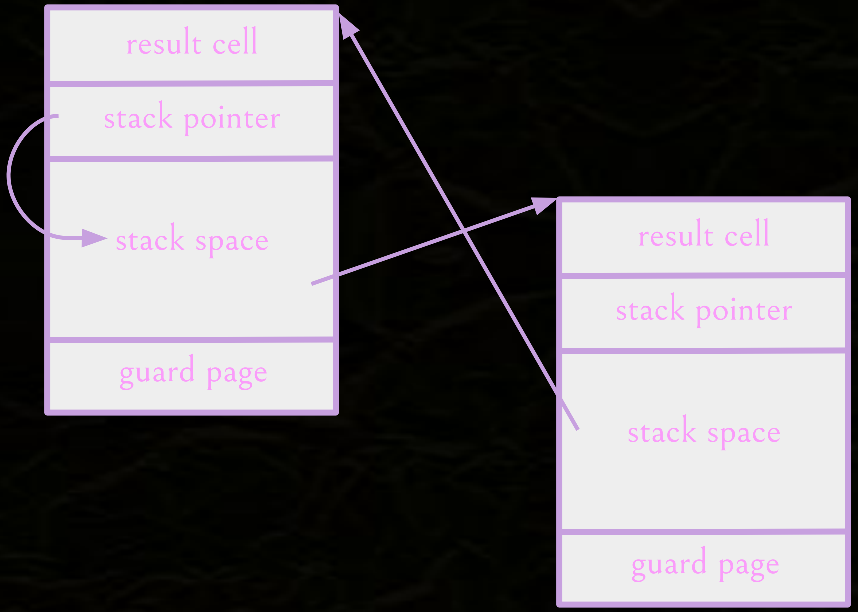
stack space

guard page

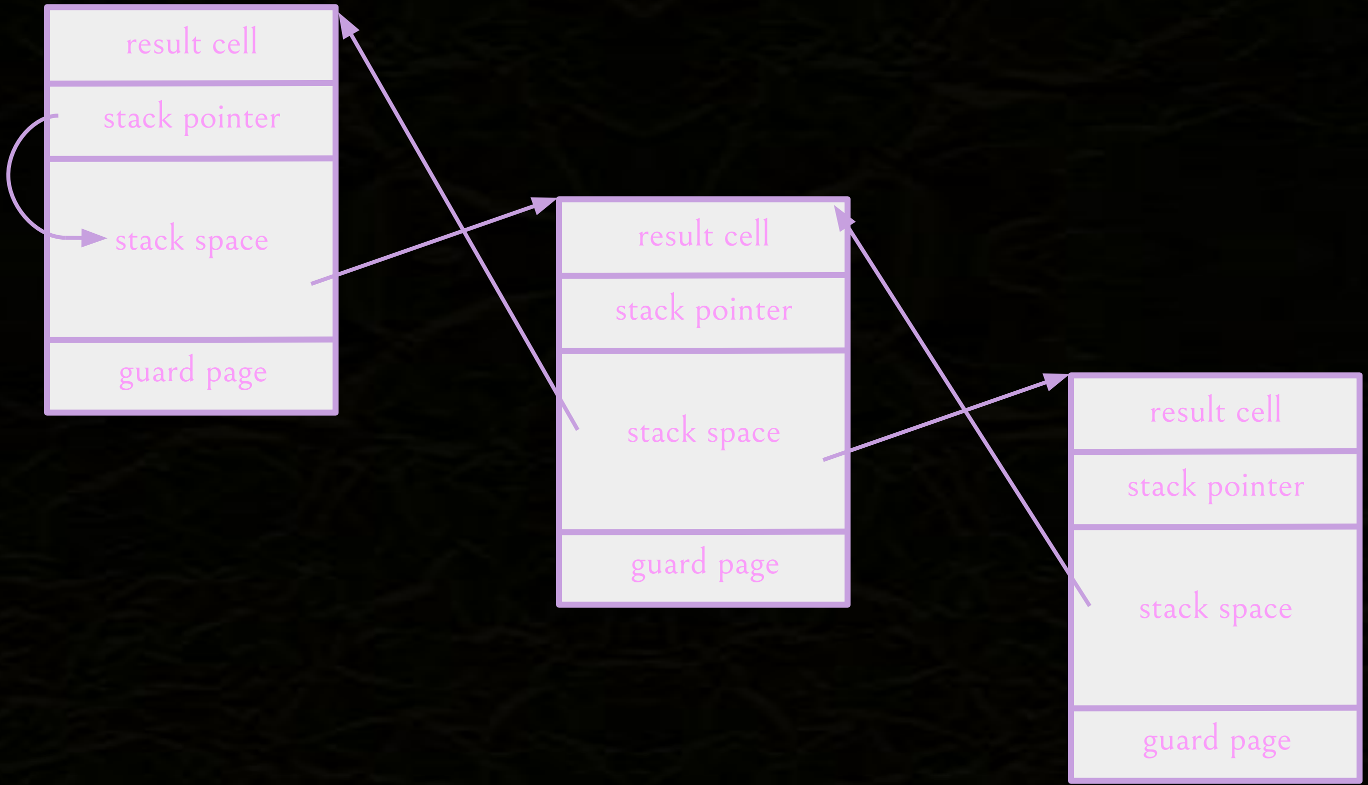
Wasmtime Fibers: Create fiber



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.new - *is it hard?*

It could be...

Currently provide fixed-size stacks

- libmprompt would provide growable stacks!

Currently provide no garbage collection - leak them!

cont.suspend - *is it hard?*

Not really!

- Fiber’s suspend functionality requires a pointer to our parent stack
- We need to keep track of who our parent is so we can suspend to them
- We keep a reference to the current stack in the context
- Maintain a stack’s parent at the top of the stack
    - Requires adjusting x86-64 linux assembly
- Point of interest: Wasmtime maintains a stack limit for safety, which needs to be adjusted

cont.resume - *is it hard?*

Yes!

Fibers / mprompt provides one handler to suspend to, but we need to find our tag!

- Suspend provides the tag index, we desugar to br_table (easy!)
- Completion gives a special sentinel value (easy!)
- 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

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

cont.bind - *is it hard?*

Don't know yet!

Probably easy to move values around in the allocated box

Could allocate space on system stack for values

resume_throw - *is it hard?*

Don't know yet!

Should be just a special resume!

barrier - *is it hard?*

Don't know yet!

Should be just a special resume with catch-all trap!

Compiling WasmFX - *is it hard?*

Dependencies:

- Function references: YES!
- Exceptions: NO!

Instructions:

- cont.new: IT COULD BE...
- suspend: NOT REALLY!
- resume: YES!
- cont.bind: DON'T KNOW YET!
- resume_throw: DON'T KNOW YET!
- barrier: DON'T KNOW YET!

github.com/effect-handlers

wasmfx.dev

Next up: benchmarking

THANK YOU!