Stack Switching

Exploring the Design Space

Andreas Rossberg 2022/10/27

Typed Continuations Proposal

Arjun Guha, Daniel Hillerström, Daan Leijen, Sam Lindley, Luna Phipps-Costin, Matija Pretnar, KC Sivaramakrishnan

Still very actively being worked on!

… Daniel & Luna implementing in Wasmtime, first prototype almost complete

… Daniel's talk on Tuesday

Design has been fairly stable, a collection of non-trivial examples have been written

We have focussed on creating the basis for actual evaluation and tangible data

… which takes time, e.g., needed to implement prerequisites like funcrefs first

Have been rather bad at PR and giving updates :)

Convergence so far

Delimited Continuations

all current suggestions implement a form of *delimited continuations*

… hierarchical stacks

… structured, clean interop with exceptions and typing

… more powerful than undelimited continuations

… in practice, typically need at least an outermost delimiter

e.g., even around threads, to guard event loop in the browser, REPL in an interpreter, …

… proven in Wasmtime fibers, Wasm/k

… what many other communities have converged on as well

Interlude: Terminology

many competing or overlapping uses of terminology

… e.g., Java 19 *continuation* ≈ Wasmtime *fiber*

sometimes reflect semantic vs implementation point of views

trying to stick to:

computation: sequential execution of a subprogram that can be suspended

continuation: remainder of a computation when suspended

stack: system resource to run a computation on

note that fiber is often used interchangeably with all three, so avoiding it for this presentation

Core Primitives

three main primitives

create : (**ref** $func) → (**ref** $suspension)

resume : (**ref** $suspension) arg* → res* ;; plus handler

suspend : (**ref** $target) arg* → res*

computations are expressed by an initial function

… this should be a perfectly ordinary function

Handlers

every **resume** is associated with a handler

basically a jump target or jump table for suspension events

mainly superficial syntactic differences in latest proposals

```
(block $l
    …
    (resume (event $tag $l))
    …
)
…  ;; event landing pad
```

```
(handler
    …
    (resume)
    …
    (event $tag
        …  ;; event landing pad
    )
)
```

The design space from here on

Main Dimensions

typing of values

handler selection

switching mechanisms

Additional Considerations

memory management

cancellation and clean-up

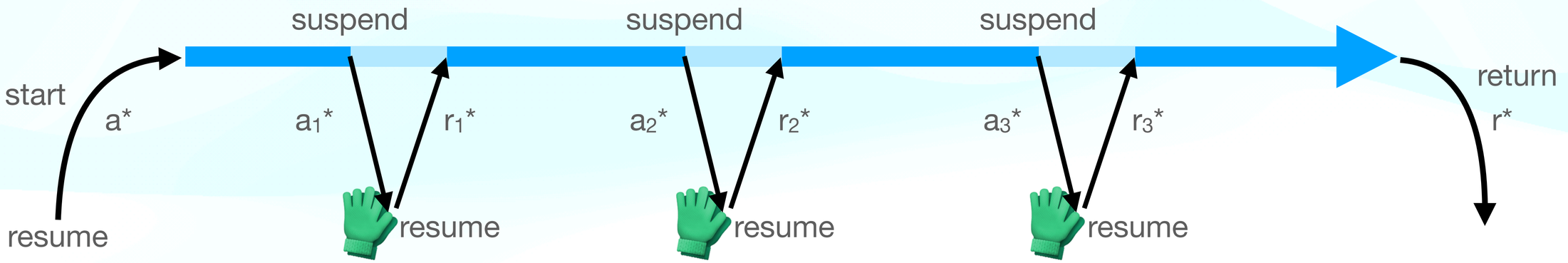
performance overhead

implementation complexity

capabilities?

Typing

Lifetime of a Computation



suspends are dealt with by a handler (a.k.a. prompt)

but what are the types of a* and r*?

… a spectrum of options with increasing precision, decreasing overhead

… in all relevant use cases, types are determined by event (event emit ≈ function call)

Event Examples

threads

```
yield : () → ()
spawn : (func () ()) → i32
self : () → i32
```

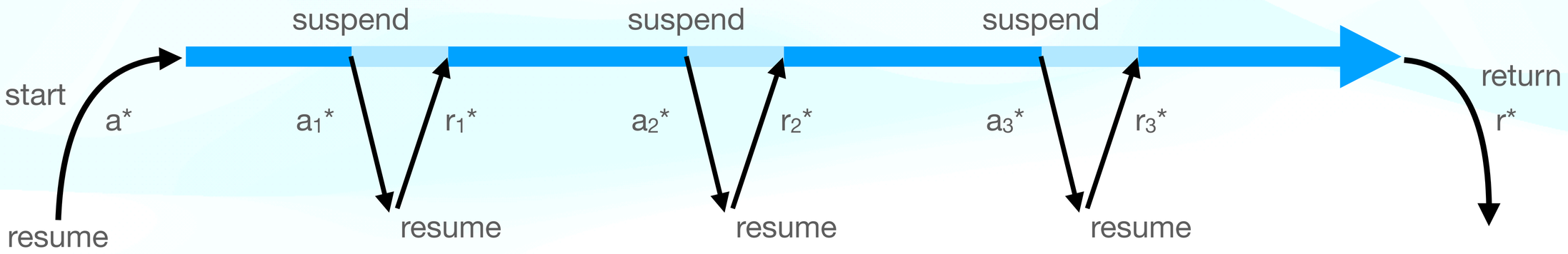
async/await

```
await : (ref $fut) → anyref
async : (func () anyref) → (ref $fut)
```

actors

```
send : (ref $actor) anyref → ()
recv : (ref $actor) → anyref
self : () → (ref $actor)
new : (func () ()) → (ref $actor)
```

Option 0: no values



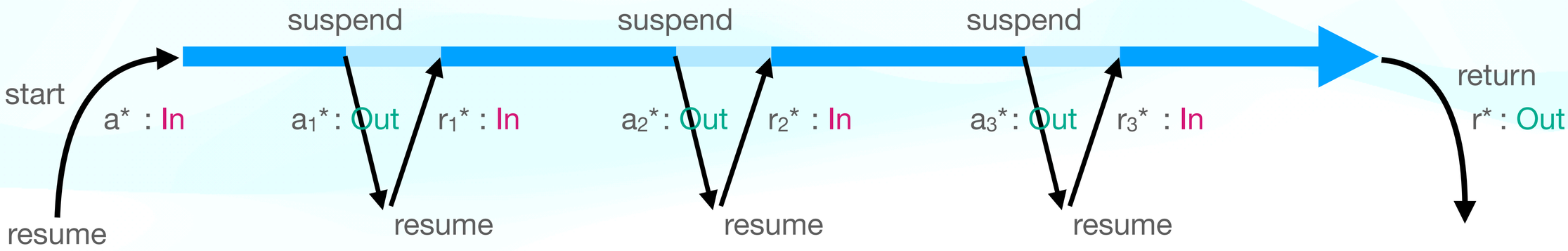
no values can be passed along resume/suspend

… e.g., traditional coroutines, Java 19

parameters and results must be funnelled through global mutable state

… undesirable for the abstractions we are interested in; problematic with threading

Option 1: homogeneous types



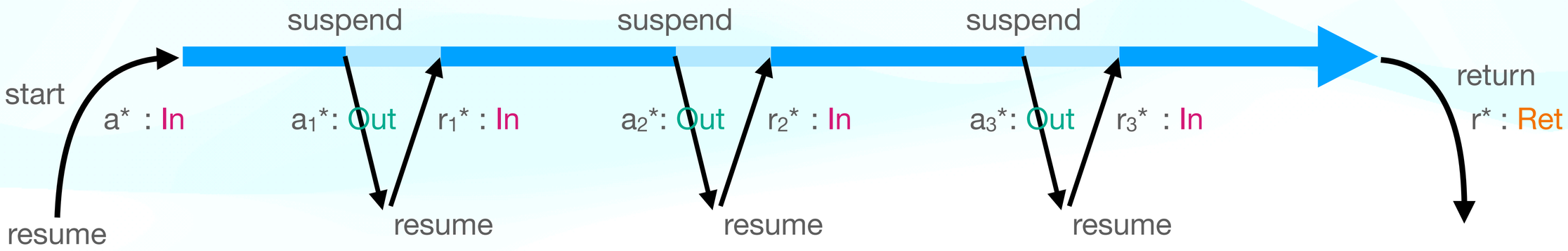
all a* and all r* have the same type

… e.g., classic shift/reset or control/prompt in untyped languages; Francis' second proposal

in practice, parameters must encode a union

… difficult and inefficient with Wasm value types; often implies boxing; non-nullable refs

Option 1b: mostly homogeneous types



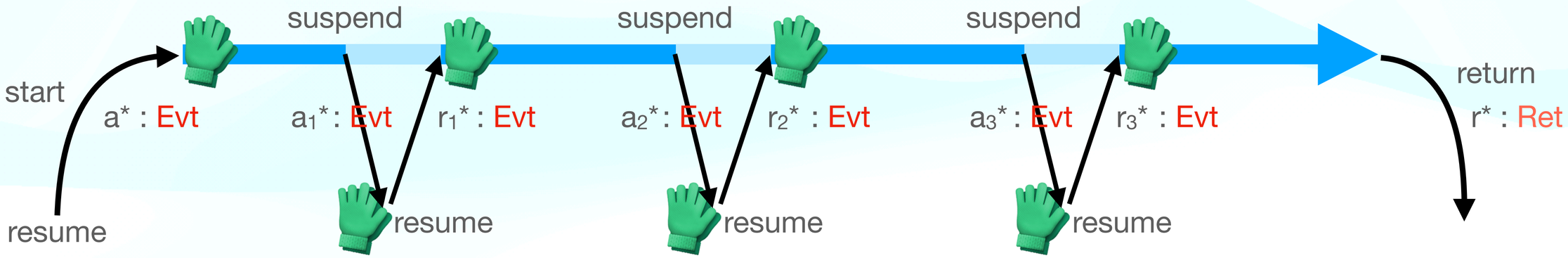
all a* and all r_i* have the same type, but r* is typed separately

… e.g., Wasmtime fibers

in practice, parameters must still encode a union

… difficult and inefficient with Wasm value types; often implies boxing; non-nullable refs

Option 2: dynamic types



all values have same/no static type but can have different dynamic type

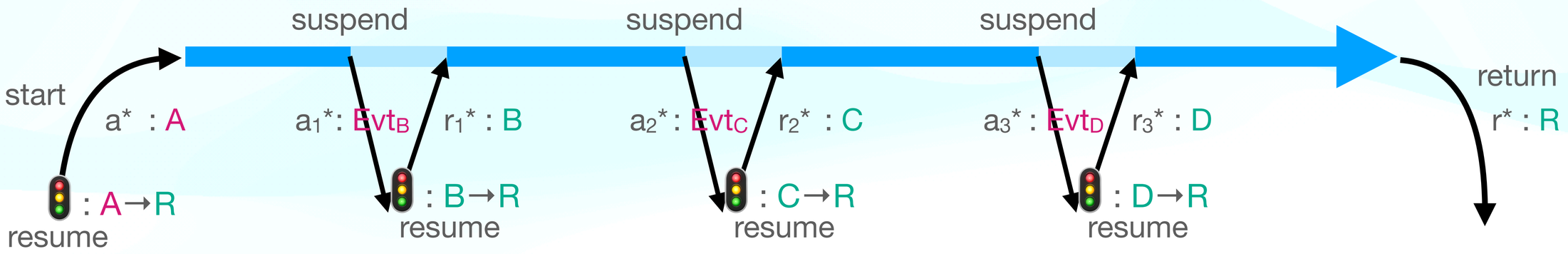
… Francis' third proposal; no precedent

requires dynamic check/dispatch/handler everywhere (start, suspend, resume)

… runtime cost; substantial complexity (e.g., *every* function needs an extra entry point)

… not a natural fit for function-like nature of events; entry handler for computation is unorthodox

Option 3: heterogeneous types



all values can have different static type, each a_i^* type implies respective r_i^* type

… e.g., typed continuations proposal; effect handlers

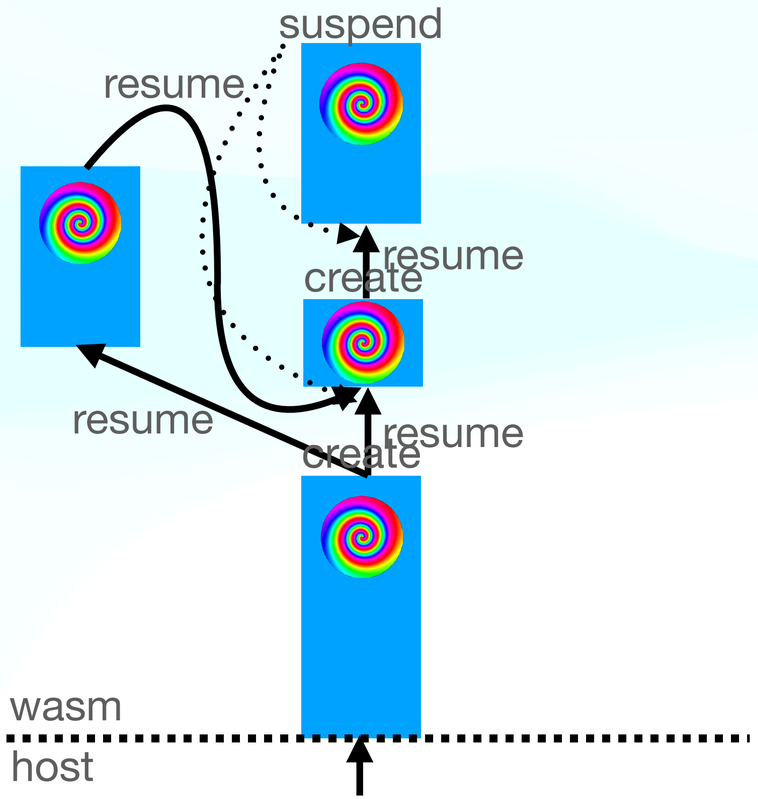
requires refining fibers to explicit continuations to distinguish resumption types

… check needed to ensure they are used only once

… but no allocation needed for them (this was a valid concern!)

Handler Selection

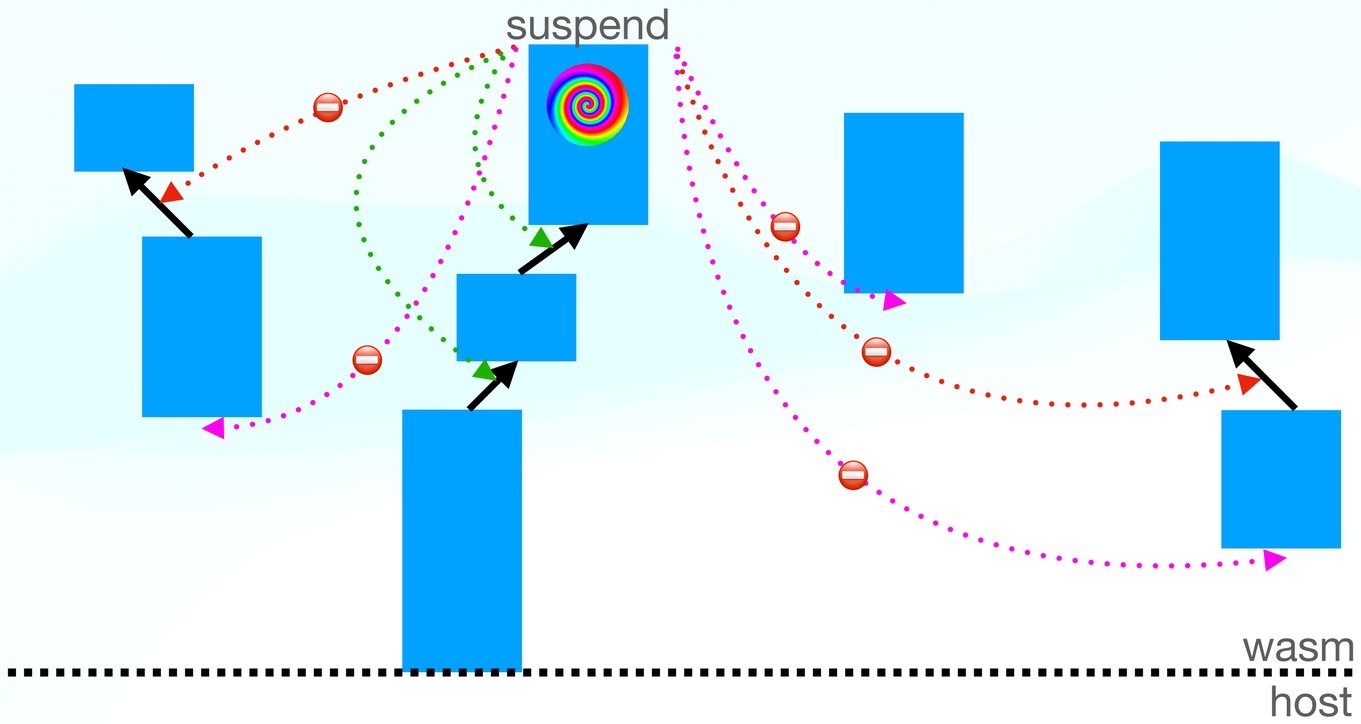
Topology of Stacks and Handlers



suspend transfers control to an active handler (arrows correspond to handlers, active when connected)

resume transfers control to a continuation (an unconnected stack and its children)

Topology of Stacks and Handlers



(1) suspension target is a handler, not a stack

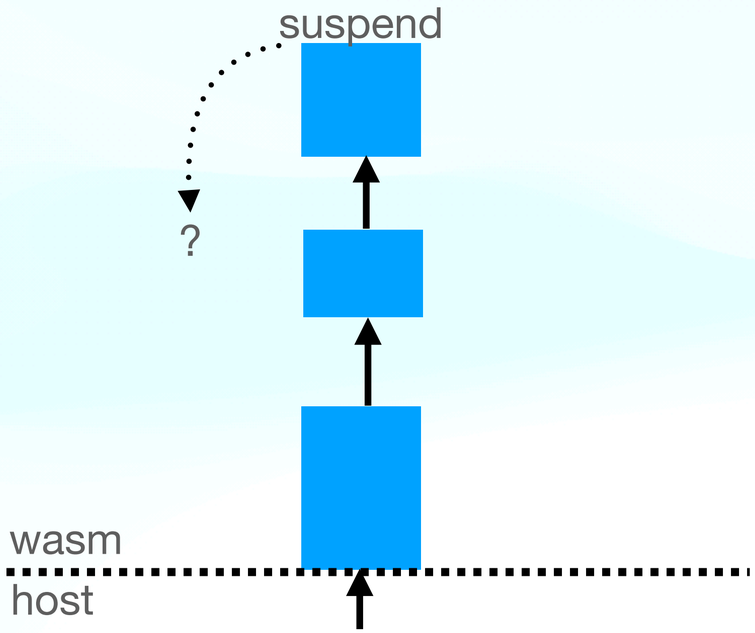
(2) suspension is only legal to handlers currently in (dynamic) scope

… handler lookup is dynamic scoping, in **all** proposals! (visible in both semantics and implementation)

… even if the “name” of the handler is known, its relative “binding” point is not

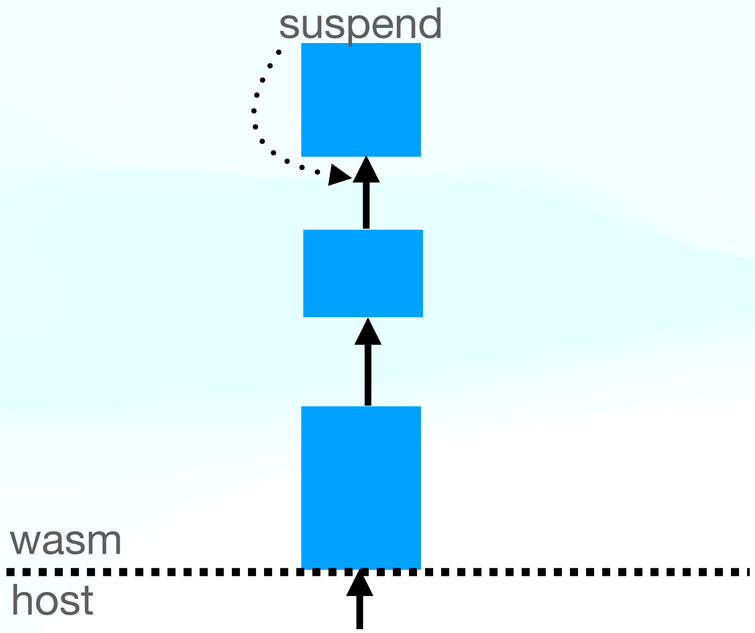
… different implementation strategies for dynamic scoping, but all have linear cost in this setting

Selecting handlers



spectrum of options for identifying a handler

Option 1: innermost handler

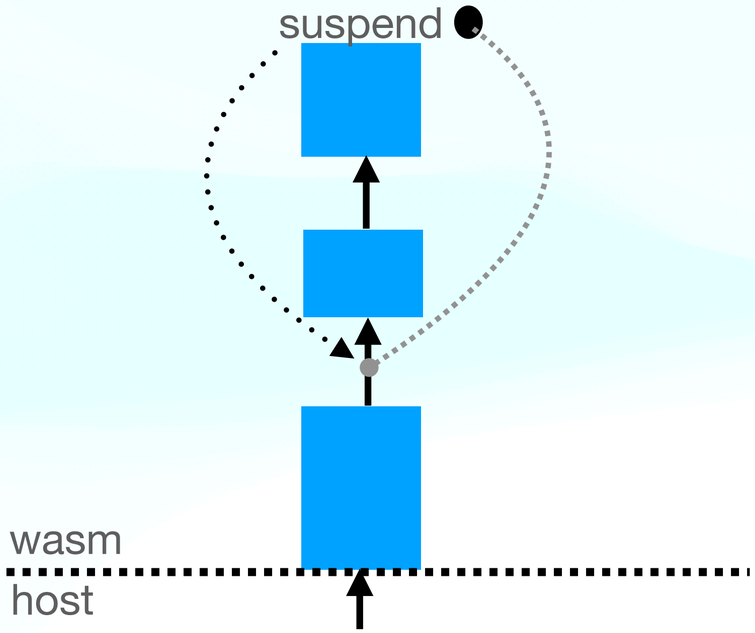


suspends unconditionally to innermost handler (a.k.a. single prompt)

… classic shift/reset or control/prompt; Wasm/k; Wasmtime fibers (?)

does not compose well for multiple different control abstractions

Option 2: by handler reference

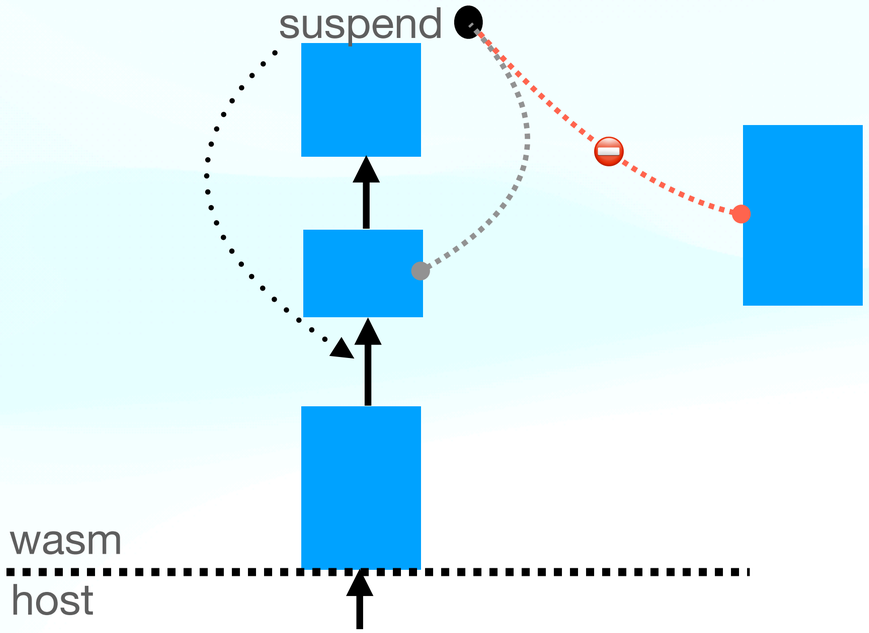


suspends to the referenced handler – if in scope

… precedent?

need to funnel reference everywhere

Option 2b: by stack reference

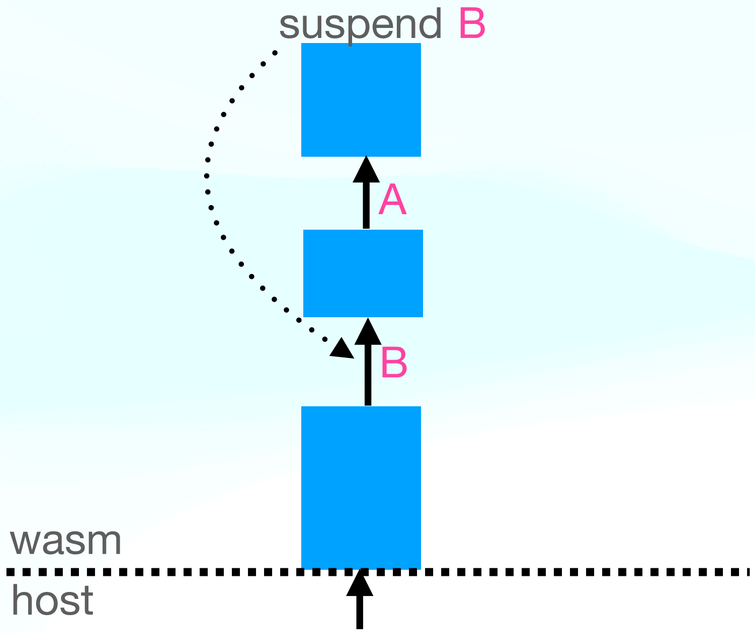


suspends to handler currently associated with respective stack

… task/fibers proposals; precedent?

have to funnel ref everywhere; additional failure cases, because stacks and handlers are not in a 1:1 relation

Option 3a: by handler label



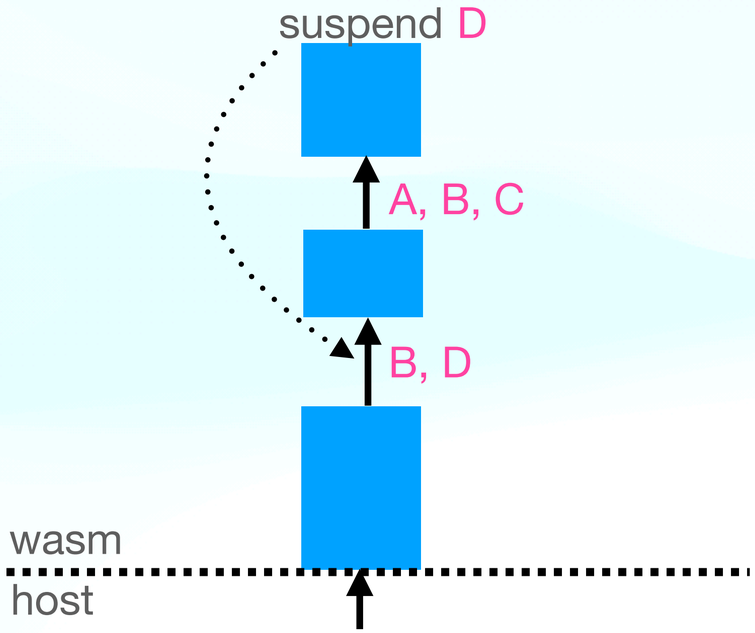
suspends to the innermost handler with respective label (a.k.a. multi prompt); labels defined separately

… Java 19 runtime; modern Scheme/Lisp libraries; libmprompt

label can be static or first-class; subsumes by-reference if labels can be first-class and generative

this is a highly relevant point in the design space, worth exploring!

Option 3b: by handler label set

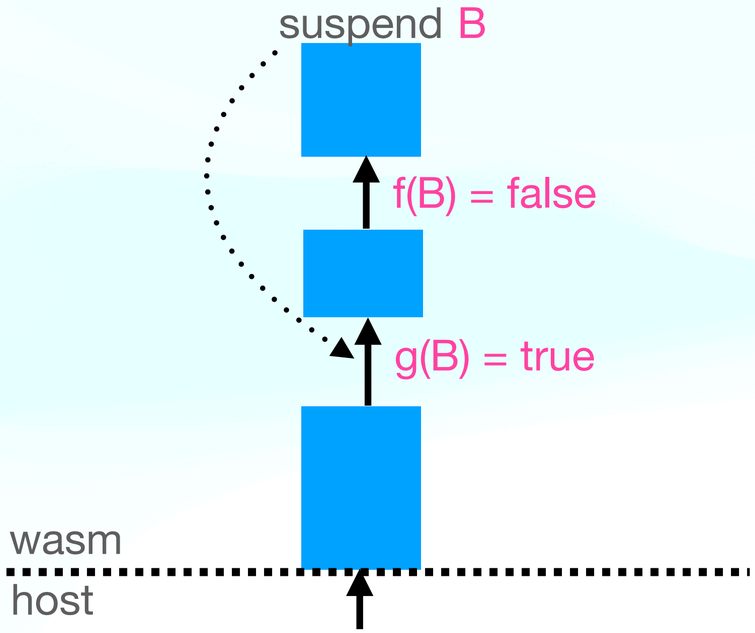


suspends to the innermost handler that includes respective label

… effect handlers; Koka; typed continuations proposal

improves composability and typing precision over single label

Option 3c: by label predicate

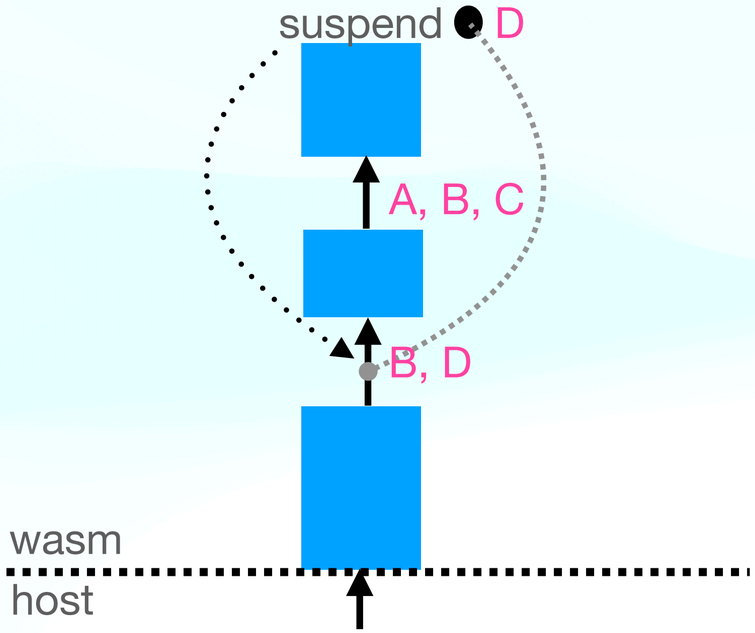


suspends to innermost handler that accepts label

… Ocaml runtime

requires closures in general; probably less optimisable

Option 1+2: by handler reference & label set



suspends to referenced handler's branch with respective label

improves static typing over mere handler reference

still have to funnel ref everywhere; superseded by first-class labels

Direct Switching

Direct switch

additional primitive, roughly, a combination of suspend+resume

switch : (to : **susp** t1* t2*) (from : **target** t3* t2*) t1* → t4*

replaces the target computation (current or parent) with another computation

motivation: bypass one hop

switch : suspend/resume ≈ tailcall : return/call

Direct switch

can be combined with any of the other choices discussed so far

complex, since it involves three stacks at the same time

“undermines” structured concurrency by unconditionally bypassing handler

… should handlers at least be able to decide whether they allow this?

unclear if it actually is a relevant performance win

premature optimisation? need to measure

Other Considerations

Memory Management

fibers depend on mutation of higher-order state and immediately allow for heap cycles

```
(type $f (func (param anyref)))
(type $s (fiber $f))

(func $f (param $x anyref) (suspend) ...)

(func $mk-cyclic-fiber (result (ref $s))
  (local $s (ref null $s))
  (local.set $s (fiber.new $s (ref.func $f)))
  (resume (local.get $s) (local.get $s))
  (local.get $s)
)
```

engines that need to prevent leaks from abandoned fibers face full GC problem

… can’t clean up after failed or abandoned modules without GC

continuations o.t.o.h. can avoid cycles, enabling RC

… implementation choice: trade-off between reusing memory and simplifying memory management

Cancellation and Clean-up

stacks may capture (runtime-internal) resources

to abort a stack, it should be properly unwound

… every stack ought to be used linearly until it either terminates or throws

simply deleting a stack is almost always wrong

… retire/release are footguns

exception mechanism already provides all the necessary functionality

Performance

We don’t know much yet, mainly guess work

Need experimental evaluation

… we have been focussing on implementation in Wasmtime lately

… complementary experiments by V8 team

Should avoid premature optimisation in the design at this stage

Capabilities?

“What is your unit of isolation?” – Mark Miller

proper capability for resume is a continuation

proper capability for suspend is a handler reference or a first-class tag

stacks or fibers are *not* capability-oriented

- … too coarse-grained: don’t distinguish suspend (≈ receive) from resume (≈ send)
- … too long-lived: once given, can be used arbitrarily later, for the whole lifetime

Use Cases

Use Cases

	events	handlers	cancellation policy
generators	1, homogeneous	local, nested	unwind
async/await	1+, heterogeneous	varies (event loop, micro events, other future notions)	unwind or other
threads	2+, heterogeneous	global	unwind
actors	3+, heterogeneous	global	none?
control/prompt	1, mostly homogeneous	local, nested	unwind+
call/cc	1, homogeneous	global	?
effect handlers	N, heterogeneous	local, nested	unwind

Summary

Next Steps

Much is assumed, little is known!

Need to converge further somehow

Typed continuations proposal takes cues from observable trend in this field

… undelimited continuations → delimited continuations → multi-prompt → effect handlers

(these go by different names in different communities, e.g., symmetric vs asymmetric coroutines, fibers, task scopes, etc)

We have many data points from other languages, but not much for Wasm

Our current focus needs to be experimentation and evaluation

… of both performance and usability

Backup

Representation of Continuations

fiber = stack + cpu state + parent fiber + i64 sequence counter

continuation = fiber + i64 sequence counter

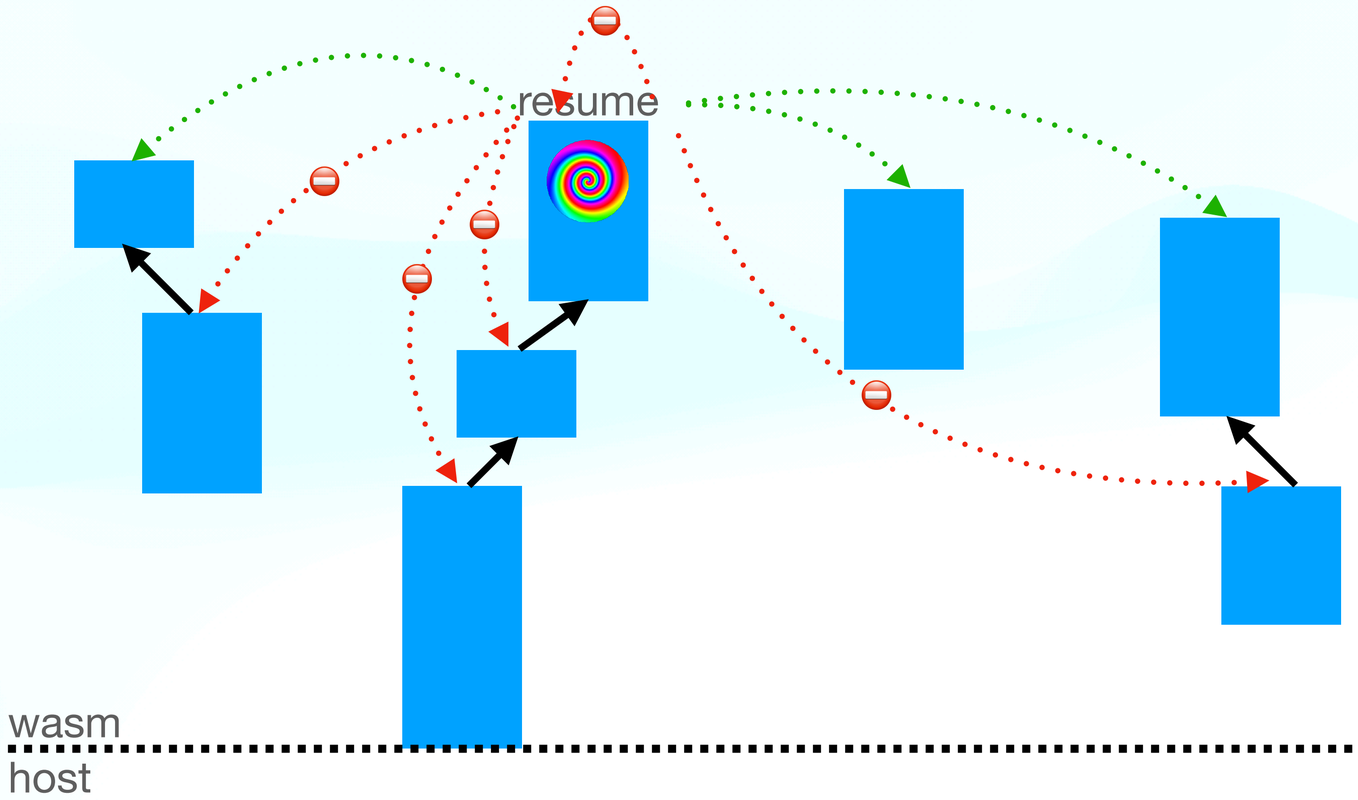
resume : (ref $continuation) args* → res*

… check that sequence counter matches up
… increment sequence counter in fiber

a continuation can be represented as a fat pointer, no allocations required

… natural choice now that we have multiple reference hierarchies

Topology of Stacks and Handlers



(1) resumption target is a continuation, encapsulating a (disconnected) *chain* of stacks, not a single stack

(2) execution proceeds on the top stack, not the disconnected bottom stack

… resume(fiber) and switch(fiber) are misleading, as they generally switch to a child of the fiber

The Problem with by-reference

various control abstractions can be nested

… generators, control/prompt, some forms of async/await, effect handlers, exception handlers, …

when suspending, need to get hold of the current handler reference for the event at hand

naturally a problem of dynamic scoping

… which the engine already implements!

… but asking for a reference asks producer to replicate that mechanism in user space

… language with N control effects generally requires N dynamically scoped bindings

Replicating Dynamic Scoping in User Space

pure approach: add N pervasive parameters to calling conventions

… or one N-tuple, but that trades off more parameters for more allocations

… significant call overhead that affects *all* functions!
(unless source language has a static type & effect system to distinguish, e.g., Koka)

impure approach: maintain a global shadow stack – actually a *forest* of stacks

… complex and brittle: have to keep two dynamic scoping mechanisms in sync

… global state state interacts poorly with other effects like threads and exceptions

… unnecessarily duplicates work the engine already does

… *did anybody ever try this*?

neither approach is cheap, neither is composable!

Selecting handlers by reference

As far as we can tell, this approach is

… complicated & error-prone

… monolithic and not composable

… more expensive than necessary

… unproven, negating the experience from other language communities

… without tangible advantage

Thank you