Proposal

Typed continuations as first-class stacks

Daniel Hillerström, Daan Leijen, Sam Lindley, Matija Pretnar, Andreas Rossberg, KC Sivaramakrishnan

stack switching

Wasm needs the ability to express control flow operators

lightweight ("green") threads
coroutines
async/await
generators
call/cc
delimited continuations
...

challenges

switch stacks at any depth

pass values between suspend/resume (both ways)

types that support validation and minimise runtime checks

composable use of multiple control operators

avoid the need for GC (no cycles)

assume heterogeneous stacks (no copying or moving frames)

proposal

typed view of stacks as first-class continuations

adopt a low-level version of effect handlers

generalise exceptions to events by adding return types

expressive mechanism, both universal and efficient

backed by well-understood semantics and PL research a reference type of suspended stacks (continuations)

(cont \$ftype)   ;; (type \$ftype (func [t_1*] → [t_2*]))

create a suspended stack, executing a function

cont.new : [(ref \$ftype)] → [(cont \$ftype)]

resume a stack, passing in expected values

cont.resume : [t_1* (cont \$ftype)] → [t_2*]

(to be refined on next slide)

“exceptions” with return values (events)

(**event** $evt (param t_P*) (result t_R*))

suspend the current stack, signalling an event to parent

cont.suspend $evt : [$t_p$*] → [$t_R$*]

resume a stack, passing in expected values, handle events

cont.resume ($evt $l)* : [$t_1$* (cont $ft)] → [$t_2$*]

handler labels $l* receive event args and next continuation

intuition

execution can suspend by emitting an event

up to program context how to handle specific events

each control operator represented by an event (/set)

decouples implementation of different operators

asymmetry enables composition through nesting finalize a stack, by injecting an exception

cont.throw $exn : [$t_E$* (cont $ftype)] → [$t_2$*]

example: green threads

```
(event $yield)
(event $spawn (param (ref $proc)))

(type $proc (func))
```

```
(event $yield)
(event $spawn (param (ref $proc)))


(func $scheduler (param $main (ref $proc))
  (cont.new (local.get $main)) (call $enqueue)
  (loop $l
    (if (call $queue_empty) (then (return)))
    (block $on_yield (result (cont $proc))
      (block $on_spawn (result (ref $proc) (cont $proc))
        (call $dequeue)
        (cont.resume ($yield $on_yield) ($spawn $on_spawn))
        (br $l)                                ;; thread terminated
      )
      ;; on $spawn, proc and cont on stack
      (call $enqueue)                          ;; continuation of old thread
      (cont.new) (call $enqueue)               ;; new thread
      (br $l)
    )
    ;; on $yield, cont on stack
    (call $enqueue)
    (br $l)
  )
)
```

```
(global $queue (list-of (cont $proc)) …)

(func $enqueue (param (cont $proc)) …)
(func $dequeue (result (cont $proc)) …)
(func $queue_empty (result i32) …)
```

example: simple generator

```
(event $enum-yield (param i64) (result i32))

(func $enum-until (param $b i32)
  (local $n i64)
  (local.set $n (i64.const  -1))
  (br_if 0 (local.get $b))
  (loop $l
    (local.set $n (i64.add (local.get $n) (i64.const 1)))
    (cont.suspend $enum-yield (local.get $n))
    (br_if $l)
  )
)
```

```
(func $run-upto (param $max i64)
  (local $n i64)
  (local $cont (cont (param i32)))
  (local.set $cont (cont.new $enum-until))
  (loop $l
    (block $h (result i64 (cont (param i32)))
      (cont.resume ($enum-yield $h)
        (i64.ge_u (local.get $n) (local.get $max))
        (local.get $cont)
      )
      (return)
    )
    (local.set $cont)
    (local.set $n)
    ;; ...process $n...
    (br $l)
  )
)
```

composition

multiple control operators can be nested

- a scheduler
  - running a thread
    - running a generator
      - calling a compute function
        - yielding to scheduler

requires no coordination between operators,
compute function can yield directly to scheduler

```
;; Compute Library


(func $compute (param …) (result …)
  …
  (loop $l


    …
    (br_if $l …)
  )
  …
)
```

```
;; Compute Library, threads-aware

(event $yield (import “threading” “yield”))

(func $compute (param …) (result …)
  …
  (loop $l
    ;; long running loop; make sure not to starve other threads
    (if (some-metric) (then (cont.suspend $yield)))
    …
    (br_if $l …)
  )
  …
)
```