Stacks and Continuations for Wasm

Idea sketch

Andreas Rossberg

Dfinity

with Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, Sam Lindley, Stephen Dolan

stack switching

support compilation of control abstraction like coroutines, generators, lightweight threads, async/await...

stack = delimited continuation

for Wasm, needs to be typed

initial idea

extend exceptions with resumption

also known as effect handlers

exnref carries continuation

```
(exception $e (param t*))
(throw $e)                    :  [t*] → ⊥
(rethrow)                     :  [exnref] → ⊥
(try (param $t_1$*) (result $t_2$*)
  …                           :  [$t_1$*] → [$t_2$*]
 catch
  …                           :  [exnref] → [$t_2$*]
)
(br_on_exn $l $e)             :  [exnref] → [exnref]
                                 iff $e : [t*] and $l : [t*]
```

```
(exception $e (param t*))
(throw $e)                      :  [t*] → ⊥
(rethrow)                       :  [exnref] → ⊥
(try (param t₁*) (result t₂*)
  …                             :  [t₁*] → [t₂*]
 catch
  …                             :  [exnref] → [t₂*]
)
(br_on_exn $l $e)               :  [exnref] → [exnref]
                                   iff $e : [t*] and $l : [t*]
```

```
(exception $e (param t*) (result t′*))
(throw $e)                              :  [t*] → ⊥
(rethrow)                               :  [exnref] → ⊥
(try (param t_1*) (result t_2*)
   …                                    :  [t_1*] → [t_2*]
 catch
   …                                    :  [exnref] → [t_2*]
)
(br_on_exn $l $e)                       :  [exnref] → [exnref]
                                           iff $e : [t*] and $l : [t*]
```

(**exception** $e (param t*) (result t′*))

(**throw** $e) : [t*] → [t′*]

(**rethrow**) : [exnref] → ⊥

(**try** (param t_1*) (result t_2*)

… : [t_1*] → [t_2*]

catch

… : [exnref] → [t_2*]

)

(**br_on_exn** $l $e) : [exnref] → [exnref]
iff $e : [t*] and $l : [t*]

(**resume**) : [(cont t′* t_2*) t′*] → [t_2*]

```
(exception $e (param t*) (result t′*))
(throw $e)                 :  [t*] → [t′*]
(rethrow)                  :  [exnref] → ⊥
(try (param t_1*) (result t_2*)
                           :  [t_1*] → [t_2*]
   …
 catch
                           :  [exnref] → [t_2*]
   …
)
(br_on_exn $l $e)          :  [exnref] → [exnref]
                              iff $e : [t*] → [t′*]
                              and $l : [t* (cont t′* t_2*)]
(resume)                   :  [(cont t′* t_2*) t′*] → [t_2*]
```

```
(exception $yield (param i32) (result i32))

 (func $gen
    (local $n i32)
    (local.set $n (i32.const -1))
    (loop $l
       (local.set $n (i32.add (local.get $n) (i32.const 1)))
       (throw $yield (local.get $n))
       (br_if $l)
    )
 )
```

```
(func $run (param $max i32)
  (local $n i32)
  (local $c (cont (param i32) (result i32)))
  (try
     (call $gen)
   catch
     (block (br_on_exn 0 $yield) (rethrow))
     (local.set $cont) (local.set $n)
     …process $n…
     (resume (local.get $cont) (i32.lt_u (local.get $n) (local.get $max)))
  )
)
```

effect handlers

entering a **try** creates a new stack

exiting the **try** regularly ends the lifetime of this stack

throw and **resume** switch between stacks

continuations are single-shot

cont is a value type, so can be stored away
…allows deferring resumption, e.g. coroutines

omitted lots of details, e.g. annotating resumability

problems

complex monolithic try instruction with obscure cost

deep vs shallow handler semantics

resumability annotations separate it from exceptions

simplify

decompose (resumable) try

replace with explicit instructions for continuations

more like (asymmetric) coroutines

$(\textbf{event}\ \$e\ (\text{param}\ t_P{*})\ (\text{result}\ t_R{*}))$

$(\textbf{cont.new}) \quad : \quad [(\text{ref}\ \$ft)] \rightarrow [(\text{cont}\ \$ft)]$

$\text{iff type}\ \$ft = [t_1{*}] \rightarrow [t_2{*}]$

$(\textbf{cont.resume}\ \$l) \quad : \quad [(\text{cont}\ \$ft)\ t_1{*}] \rightarrow [t_2{*}]$

$\text{iff label}\ \$l : [(\text{evtref}\ t_2{*})]$

$(\textbf{cont.yield}\ \$e) \quad : \quad [t_P{*}] \rightarrow [t_R{*}]$

$\text{iff event}\ \$e : [t_P{*}] \rightarrow [t_R{*}]$

$(\textbf{br\_on\_evt}\ \$l\ \$e) \quad : \quad [(\text{evtref}\ t_2{*})] \rightarrow [(\text{evtref}\ t_2{*})]$

$\text{iff event}\ \$e : [t_P{*}] \rightarrow [t_R{*}]$

$\text{and label}\ \$l : [t_P{*}\ (\text{cont}\ \$ft')]$

$\text{and type}\ \$ft' = [t_R{*}] \rightarrow [t_2{*}]$

new types

continuations (cont \$ft) where type \$ft = $[t_1{*}] \rightarrow [t_2{*}]$

… a "coroutine" whose resumption needs t_1* values
and that will terminate with t_2*

events (evtref t*)

… a packet of an event id, its argument values,
and a continuation that terminates with t*
(like exnref with a continuation)

$(\textbf{event}\ \$e\ (\text{param}\ t_P{*})\ (\text{result}\ t_R{*}))$

$(\textbf{cont.new}) \quad : \quad [(\text{ref}\ \$ft)] \rightarrow [(\text{cont}\ \$ft)]$
$\quad \text{iff type}\ \$ft = [t_1{*}] \rightarrow [t_2{*}]$

$(\textbf{cont.resume}\ \$l) \quad : \quad [(\text{cont}\ \$ft)\ t_1{*}] \rightarrow [t_2{*}]$
$\quad \text{iff label}\ \$l : [(\text{evtref}\ t_2{*})]$

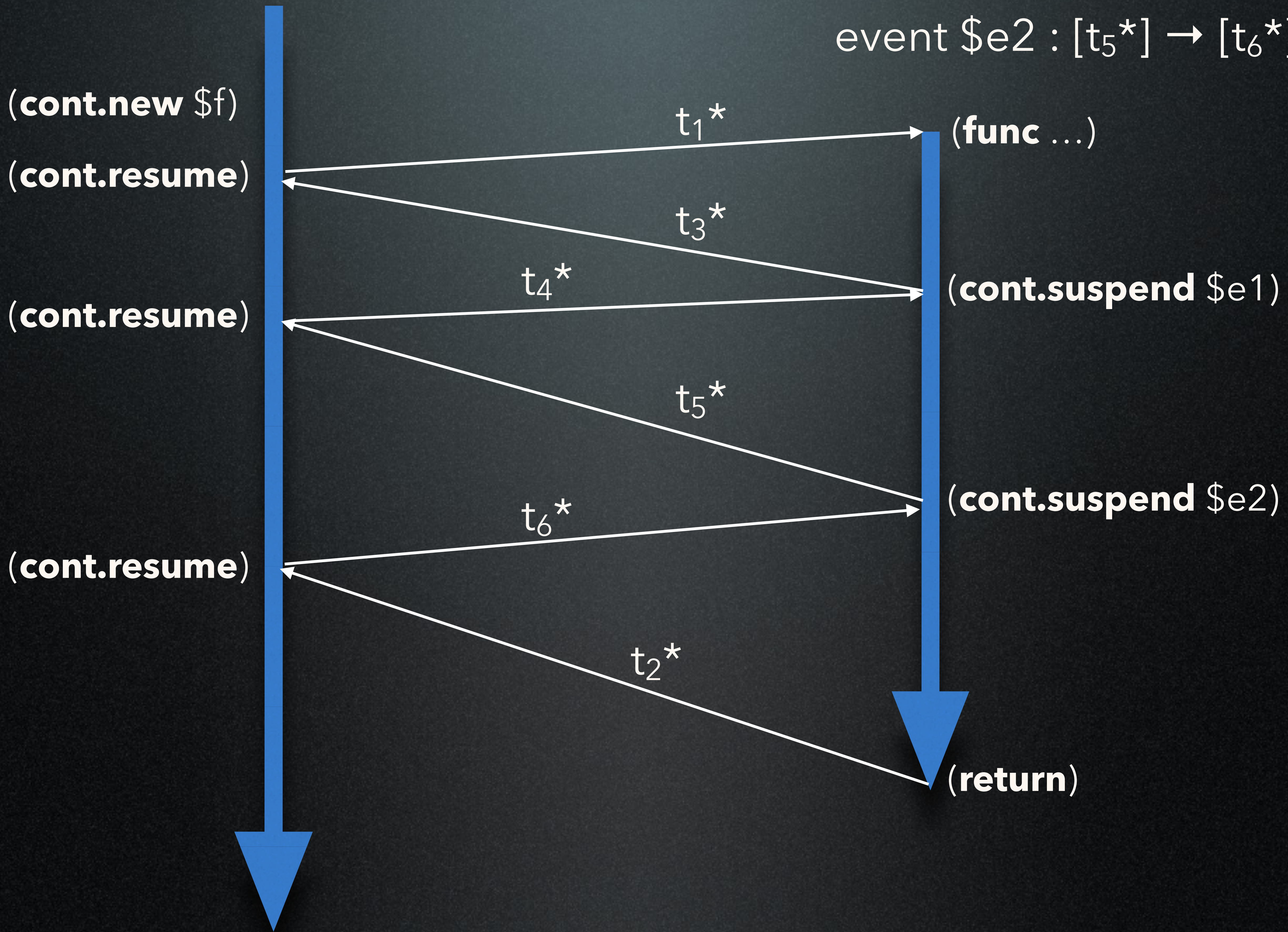
$(\textbf{cont.yield}\ \$e) \quad : \quad [t_P{*}] \rightarrow [t_R{*}]$
$\quad \text{iff event}\ \$e : [t_P{*}] \rightarrow [t_R{*}]$

$(\textbf{br\_on\_evt}\ \$l\ \$e) \quad : \quad [(\text{evtref}\ t_2{*})] \rightarrow [(\text{evtref}\ t_2{*})]$
$\quad \text{iff event}\ \$e : [t_P{*}] \rightarrow [t_R{*}]$
$\quad \text{and label}\ \$l : [t_P{*}\ (\text{cont}\ \$ft')]$
$\quad \text{and type}\ \$ft' = [t_R{*}] \rightarrow [t_2{*}]$

func $f : [$t_1$*] → [$t_2$*]

event $e1 : [$t_3$*] → [$t_4$*]

event $e2 : [$t_5$*] → [$t_6$*]



example: thread scheduler

```
(type $proc (func))

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

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

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

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

(func $scheduler (param $main (ref $proc))
  (call $enqueue (cont.new (local.get $main)))
  (loop $l
    (if (i32.eqz (call $queued)) (then (return)))
    (block $on_event (result (evtref))
      (cont.resume $on_event (call $dequeue))
      (br $l)
    )
    (switch-on-evtref
      (case $yield                                ;; cont on stack
          (call $enqueue))
      (case $fork                                 ;; proc and cont on stack
          (cont.new) (call $enqueue)
          (call $enqueue))
      (default                                    ;; evtref on stack
          (cont.reyield))
    )
    (br $l)
  )
)
```

$(\textbf{event}\ \$e\ (\text{param}\ t_P{*})\ (\text{result}\ t_R{*}))$

$(\textbf{cont.new}) \quad : \quad [(\text{ref}\ \$ft)] \rightarrow [(\text{cont}\ \$ft)]$
$\text{iff type}\ \$ft = [t_1{*}] \rightarrow [t_2{*}]$

$(\textbf{cont.resume}\ \$l) \quad : \quad [(\text{cont}\ \$ft)\ t_1{*}] \rightarrow [t_2{*}]$
$\text{iff label}\ \$l : [(\text{evtref}\ t_2{*})]$

$(\textbf{cont.yield}\ \$e) \quad : \quad [t_P{*}] \rightarrow [t_R{*}]$
$\text{iff event}\ \$e : [t_P{*}] \rightarrow [t_R{*}]$

$(\textbf{br\_on\_evt}\ \$l\ \$e) \quad : \quad [(\text{evtref}\ t_2{*})] \rightarrow [(\text{evtref}\ t_2{*})]$
$\text{iff event}\ \$e : [t_P{*}] \rightarrow [t_R{*}]$
$\text{and label}\ \$l : [t_P{*}\ (\text{cont}\ \$ft')]$
$\text{and type}\ \$ft' = [t_R{*}] \rightarrow [t_2{*}]$

(**event** \$e (param t_P*) (result t_R*))

(**cont.new**) : $[(\mathrm{ref}\ \$ft)] \to [(\mathrm{cont}\ \$ft)]$
iff type \$ft = $[t_1{*}] \to [t_2{*}]$

(**cont.resume** \$l) : $[(\mathrm{cont}\ \$ft)\ t_1{*}] \to [t_2{*}]$
iff label \$l : $[(\mathrm{evtref}\ t_2{*})]$

(**cont.yield** \$e) : $[t_P{*}] \to [t_R{*}]$
iff event \$e : $[t_P{*}] \to [t_R{*}]$

(**cont.reyield** \$l) : $[(\mathrm{evtref}\ t_2{*})] \to [t_2{*}]$
iff label \$l : $[(\mathrm{evtref}\ t_2{*})]$

(**cont.throw** \$e) : $[(\mathrm{cont}\ \$ft)\ t{*}] \to [t_2{*}]$
iff exception \$e : $[t{*}]$