WasmFX: Typed Continuations in Wasm

Daniel Hillerström [1] Sam Lindley [1]
Andreas Rossberg [2] KC Sivaramakrishnan [3] Daan Leijen [4] Matija Pretnar [5]

[1] The University of Edinburgh, UK

[2] DFINITY, CH

[3] IIT Madras, IN

[4] Microsoft Research, USA

[5] University of Ljubljana, SI

October 20, 2021

WebAssembly: neither web nor assembly (Haas et al. 2017)

What is Wasm?

- A universal compilation target
- A virtual stack machine (source language agnostic)
- A predictable performance model

Code format

- A Wasm "program" is a structured module
- Designed for stream compilation
- The term language is *statically typed* and block-structured
- Control flow is structured (*i.e.* all CFGs are reducible)

Wasm MVP 1.0 is tailored for C/C++

WasmFX extends Wasm with first-class continuations

The problem

- Non-local control flow abstractions are pervasive (*e.g.* async/await, lightweight threads, first-class continuations)
- Wasm lacks support for non-local control flow

The solution

- Handling-style delimited continuations (Sitaram (1993), Plotkin and Pretnar (2009))
- Admits easy typing using insights from effect handlers
- Minimal extension to Wasm
    - Introduction of control tags
    - A type constructor for continuations
    - Six instructions for manipulating (linear) continuations

Deep or shallow semantics?

Deep capture and resumption

$$\text{handle } \mathcal{E}[\text{op } V] \text{ with } H \rightsquigarrow N[\text{cont}_{\langle H;\mathcal{E}\rangle}/r, V/x], \text{ where } \{\text{op } p\ r \mapsto N\} \in H$$

$$\text{resume } V \text{ with } W \rightsquigarrow \text{handle } \mathcal{E}[W] \text{ with } H, \text{ where } V = \text{cont}_{\langle H;\mathcal{E}\rangle}$$

Shallow capture and resumption

$$\text{handle } \mathcal{E}[\text{op } V] \text{ with } H \rightsquigarrow N[\text{cont}_{\langle \mathcal{E}\rangle}/r, V/x], \text{ where } \{\text{op } p\ r \mapsto N\} \in H$$

$$\text{resume } V \text{ with } W \rightsquigarrow \mathcal{E}[W], \qquad \text{where } V = \text{cont}_{\langle \mathcal{E}\rangle}$$

Deep or shallow semantics?

Deep capture and resumption

$$\begin{aligned}\text{handle } \mathcal{E}[\text{op } V] \text{ with } H &\leadsto N[\text{cont}_{\langle H;\mathcal{E}\rangle}/r, V/x], \text{ where } \{\text{op } p\ r \mapsto N\} \in H \\ \text{resume } V \text{ with } W &\leadsto \text{handle } \mathcal{E}[W] \text{ with } H, \text{ where } V = \text{cont}_{\langle H;\mathcal{E}\rangle}\end{aligned}$$

Shallow capture and resumption

$$\begin{aligned}\text{handle } \mathcal{E}[\text{op } V] \text{ with } H &\leadsto N[\text{cont}_{\langle\mathcal{E}\rangle}/r, V/x], \text{ where } \{\text{op } p\ r \mapsto N\} \in H \\ \text{resume } V \text{ with } W &\leadsto \mathcal{E}[W], \qquad \text{where } V = \text{cont}_{\langle\mathcal{E}\rangle}\end{aligned}$$

'Sheep' allocation, capture, and resumption

$$\begin{aligned}\text{cont.new } V &\leadsto \text{cont}_{\langle V\rangle}, & &\text{where } V = \lambda\langle\rangle.M \\ \underline{\text{handle}}\ \mathcal{E}[\text{op } V]\ \underline{\text{with}}\ H &\leadsto N[\text{cont}_{\langle\mathcal{E}\rangle}/r, V/x], & &\text{where } \{\text{op } p\ r \mapsto N\} \in H \\ \text{resume } V \text{ with } \langle H; W\rangle &\leadsto \underline{\text{handle}}\ \mathcal{E}[W]\ \underline{\text{with}}\ H, & &\text{where } V = \text{cont}_{\langle\mathcal{E}\rangle}\end{aligned}$$

Running example: coroutines (1)

```
;; interface for running two coroutines
;; non-interleaving implementation
(module $co2
  ;; type alias task = [] -> []
  (type $task (func))

  ;; yield : [] -> []
  (func $yield (export "yield")
    (nop))

  ;; run : [(ref $task) (ref $task)] -> []
  (func $run (export "run") (param $task1 (ref $task)) (param $task2 (ref $task))
    ;; run the tasks sequentially
    (call_ref (local.get $task1))
    (call_ref (local.get $task2))
  )
)
```

Running example: coroutines (2)

```
;; main example: streams of odd and even naturals
(module $example
  ...
  ;; imports yield : [] -> []
  (func $yield (import "co2" "yield"))

  ;; odd : [i32] -> []
  ;; prints the first $niter odd natural numbers
  (func $odd (param $niter i32)
        (local $n i32) ;; next odd number
        (local $i i32) ;; iterator
        ;; initialise locals
        (local.set $n (i32.const 1))
        (local.set $i (i32.const 1))
        (block $b
         (loop $l
          (br_if $b (i32.gt_u (local.get $i) (local.get $niter)))
          ;; print the current odd number
          (call $print (local.get $n))
          ;; compute next odd number
          (local.set $n (i32.add (local.get $n) (i32.const 2)))
          ;; increment the iterator
          (local.set $i (i32.add (local.get $i) (i32.const 1)))
          ;; yield control
          (call $yield)
          (br $l))))

  ;; even : [i32] -> []
  ;; prints the first $niter even natural numbers
  (func $even (param $niter i32) ...)

  ;; odd5, even5 : [] -> []
  (func $odd5 (export "odd5")
        (call $odd (i32.const 5)))
  (func $even5 (export "even5")
        (call $even (i32.const 5)))
)
```

Instructions: declaring control tags

Control tag declaration

$$(\text{tag } \$\textit{tag } (\text{param } \sigma^*) \ (\text{result } \tau^*))$$

it's a mild extension of Wasm's *exception tags*

(known in the literature as an 'operation symbol' (Plotkin and Pretnar 2009))

Refactoring the co2 module (1)

```
(module $co2
  ;; type alias task = [] -> []
  (type $task (func))

  ;; yield : [] -> []
  (tag $yield (param) (result))

  ;; yield : [] -> []
  (func $yield (export "yield")
    (nop))

  ;; run : [(ref $task) (ref $task)] -> []
  (func $run (export "run") (param $task1 (ref $task)) (param $task2 (ref $task))
    ...)
)
```

Instructions: creating continuations

Continuation type

$$(\mathsf{cont}\ ([\sigma^*] \to [\tau^*]))$$

cont is a new reference type constructor parameterised by a function type

Continuation allocation

$$\mathsf{cont.new} : [(\mathsf{ref}\ ([\sigma^*] \to [\tau^*]))] \to [(\mathsf{cont}\ ([\sigma^*] \to [\tau^*]))]$$

where ref is the type constructor for function reference types

Refactoring the co2 module (2)

```
(module $co2
  ;; type alias task = [] -> []
  (type $task (func))
  ;; type alias   ct = $task
  (type $ct   (cont $task))

  ;; yield : [] -> []
  (tag $yield (param) (result))

  ;; yield : [] -> []
  (func $yield (export "yield")
    (nop))

  ;; run : [(ref $task) (ref $task)] -> []
  ;; implements a 'seesaw' (c.f. Ganz et al. (ICFP@99))
  (func $run (export "run") (param $task1 (ref $task)) (param $task2 (ref $task))
    ;; locals to manage continuations
    (local $up    (ref null $ct))
    (local $down (ref null $ct))
    (local $isOtherDone i32)
    ;; initialise locals
    (local.set $up   (cont.new (type $ct) (local.get $task1)))
    (local.set $down (cont.new (type $ct) (local.get $task2)))
    ...)
)
```

Instructions: invoking continuations

Continuation resumption

$$\text{cont.resume (tag } \$tag\ \$h)^* : [\sigma^*\ (\text{cont } ([\sigma^*] \to [\tau^*]))] \to [\tau^*]$$

where $\{\$tag : [\sigma_i^*] \to [\tau_i^*]$ and $\$h : [\sigma_i^*\ (\text{cont } [\tau_i^*] \to [\tau^*])]\}_i$

Continuation cancellation

$$\text{cont.throw (exception } \$exn) \text{ (tag } \$tag\ \$h)^* : [\sigma_0^*\ (\text{cont } ([\sigma^*] \to [\tau^*]))] \to [\tau^*]$$

where $\$exn : [\sigma_0^*] \to []$, $\{\$tag : [\sigma_i^*] \to [\tau_i^*]$ and $\$h : [\sigma_i^*\ (\text{cont } [\tau_i^*] \to [\tau^*])]\}_i$

Both instructions fully consume their continuation argument

Refactoring the co2 module (3)

```
(module $co2
  ...
  ;; run : [(ref $task) (ref $task)] -> []
  ;; implements a 'seesaw' (c.f. Ganz et al. (ICFP@99))
  (func $run (export "run") (param $task1 (ref $task)) (param $task2 (ref $task))
    ;; locals to manage continuations
    (local $up   (ref null $ct))
    (local $down (ref null $ct))
    (local $isOtherDone i32)
    ;; initialise locals
    (local.set $up   (cont.new (type $ct) (local.get $task1)))
    (local.set $down (cont.new (type $ct) (local.get $task2)))
    ;; run $up
    (loop $h
      (block $on_yield (result (ref $ct))
        (cont.resume (tag $yield $on_yield)
                     (local.get $up))
        ;; $up finished, check whether $down is done
        (if (i32.eq (local.get $isOtherDone) (i32.const 1))
          (then (return)))
        ;; prepare to run $down
        (local.get $down)
        (local.set $up)
        (local.set $isOtherDone (i32.const 1))
        (br $h)
      ) ;; on_yield clause, stack type: [(cont $ct)]
      (local.set $up)
      (if (i32.eqz (local.get $isOtherDone))
        (then
        ;; swap $up and $down
        (local.get $down)
        (local.set $down (local.get $up))
        (local.set $up)
      ))
      (br $h)))
)
```

Instructions: suspending continuations

Continuation suspension

$$\text{cont.suspend } \$\mathit{tag} : [\sigma^*] \to [\tau^*]$$

where $\$\mathit{tag} : [\sigma^*] \to [\tau^*]$

Refactoring the co2 module (4)

```
(module $co2
 ;; type alias task = [] -> []
 (type $task (func))
 ;; type alias   ct = $task
 (type $ct   (cont $task))

 ;; yield : [] -> []
 (tag $yield (param) (result))

 ;; yield : [] -> []
 (func $yield (export "yield")
   (cont.suspend $yield))

 ;; run : [(ref $task) (ref $task)] -> []
 ;; implements a 'seesaw' (c.f. Ganz et al. (ICFP@99))
 (func $run (export "run") (param $task1 (ref $task)) (param $task2 (ref $task))
  ... )
)
```

Now `(call $run (ref.func $odd5) (ref.func $even5))` prints `1 2 3 4 5 6 7 8 9 10`

Instructions: binding and trapping continuations

Partial continuation application

$$\text{cont.bind (type } \$ct) : [\sigma_0^* \text{ (cont } ([\sigma_0^* \, \sigma_1^*] \to [\tau^*]))] \to [(\text{cont } ([\sigma_1^*] \to [\tau^*]))]$$

where $\$ct = \text{cont } ([\sigma_0^* \, \sigma_1^*] \to [\tau^*])$

Control barriers

$$\text{barrier } \$lbl \text{ (type } \$bt) \; instr^* : [\sigma^*] \to [\tau^*]$$

where $\$bt = [\sigma^*] \to [\tau^*]$ and $instr^* : [\sigma^*] \to [\tau^*]$

Summary

In summary

- Typed continuations proposal adds first-class control to Wasm
- A marriage of deep and shallow handlers
- It's a minimal extension to Wasm

The proposal is being actively developed at

`https://github.com/effect-handlers/wasm-spec`

Comments and feedback are welcome!

References

Sitaram, Dorai (1993). "Handling Control". In: *PLDI*. ACM, pp. 147–155.

Ganz, Steven E., Daniel P. Friedman, and Mitchell Wand (1999). "Trampolined Style". In: *ICFP*. ACM, pp. 18–27.

Plotkin, Gordon D. and Matija Pretnar (2009). "Handlers of Algebraic Effects". In: *ESOP*. Vol. 5502. LNCS. Springer, pp. 80–94.

Haas, Andreas et al. (2017). "Bringing the web up to speed with WebAssembly". In: *PLDI*. ACM, pp. 185–200.