WasmFX: Stack Switching via Effect Handlers in WebAssembly

Daniel Hillerström

Laboratory for Foundations of Computer Science
The University of Edinburgh
Scotland, United Kingdom

January 13, 2023

I am but one of many

Sam Lindley

Andreas Rossberg

Daan Leijen

KC Sivaramakrishnan

Matija Pretnar

Luna Phipps-Costin

Arjun Guha

`https://wasmfx.dev`

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 streaming 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++

`https://webassembly.org`

The need for stack switching in Wasm

Non-local control is pervasive in programming languages

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Generators and iterators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)
- First-class continuations (e.g. Haskell, Java, OCaml, Scheme)

The need for stack switching in Wasm

Non-local control is pervasive in programming languages

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Generators and iterators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)
- First-class continuations (e.g. Haskell, Java, OCaml, Scheme)

The problem

How do I compile non-local control flow abstractions to Wasm?

The need for stack switching in Wasm

Non-local control is pervasive in programming languages

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Generators and iterators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)
- First-class continuations (e.g. Haskell, Java, OCaml, Scheme)

The problem

How do I compile non-local control flow abstractions to Wasm?

Solution

- Ceremoniously transform my entire source programs (e.g. Asyncify, CPS)

The need for stack switching in Wasm

Non-local control is pervasive in programming languages

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Generators and iterators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)
- First-class continuations (e.g. Haskell, Java, OCaml, Scheme)

The problem

How do I compile non-local control flow abstractions to Wasm?

Solution

- ~~Ceremoniously transform my entire source programs (e.g. Asyncify, CPS)~~
- Add each abstraction as a primitive to Wasm

The need for stack switching in Wasm

Non-local control is pervasive in programming languages

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Generators and iterators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)
- First-class continuations (e.g. Haskell, Java, OCaml, Scheme)

The problem

How do I compile non-local control flow abstractions to Wasm?

Solution

- ~~Ceremoniously transform my entire source programs (e.g. Asyncify, CPS)~~
- ~~Add each abstraction as a primitive to Wasm~~
- Use *effect handlers* as a unified modular basis for control in Wasm

Perspectives on effect handlers

Operational interpretation

First-class resumable exceptions

Software engineering interpretation

Composable monads builders (monads as a design pattern)

Functional programming interpretation

Folds over computation trees

Mathematical interpretation

Homomorphisms between free algebraic models

Effect handlers are a proven technology

A modular and extensible basis

- Structured form of delimited control
- Easy encoding of *your favourite abstraction* via effect handlers
- Trivially compatible with typed representations

Practical evidence

- 100+ peer reviewed papers
- Available in many programming languages (e.g. C++, Haskell, Pyro, OCaml, Unison)
- Deployed in industrial technologies (e.g. GitHub's semantic, Meta's React, Uber's Pyro)

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)

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

  ...
)
```

Running example: coroutines (3)

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

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

Running example: coroutines (4)

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


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

Instructions: declaring control tags

Control tag declaration

$$(\texttt{tag}\ \$\mathit{tag}\ (\texttt{param}\ \sigma^*)\ (\texttt{result}\ \tau^*))$$

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

(known in the literature as an 'operation symbol' (Plotkin and Pretnar 2013))

Refactoring the co2 module (1)

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

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

  ;; 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

$$(\texttt{cont}\ \$\mathit{ft})$$

cont is a new reference type constructor parameterised by a function type, $\$\mathit{ft} : [\sigma^*] \to [\tau^*]$

Continuation allocation

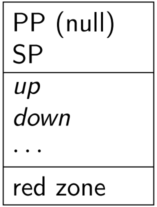
$$\texttt{cont.new} : [(\texttt{ref null}\ \$\mathit{ft})] \to [(\texttt{ref}\ \$\mathit{ct})]$$

where $\$\mathit{ft} : [\sigma^*] \to [\tau^*]$
and $\$\mathit{ct} : \texttt{cont}\ \$\mathit{ft}$

Refactoring the co2 module (2)

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

  ;; type alias $ct = $task
  (type $ct   (cont $task))

  ...

  ;; 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)))
    ...)
)
```

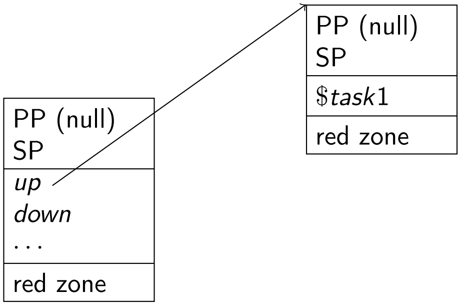
Thinking of **cont.new** in terms of stacks



cont.new allocates a new stack segment

New segments are initially suspended

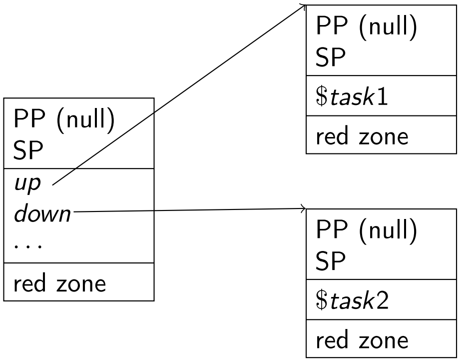
Thinking of **`cont.new`** in terms of stacks



`cont.new` allocates a new stack segment

New segments are initially suspended

Thinking of **`cont.new`** in terms of stacks



`cont.new` allocates a new stack segment

New segments are initially suspended

Instructions: invoking continuations

Continuation resumption

$$\texttt{resume}\ (\texttt{tag}\ \$tag\ \$h)^* : [\sigma^*\ (\texttt{ref null}\ \$ct)] \to [\tau^*]$$

$$\begin{array}{lllll} \text{where} & \{\$tag_i & : [\sigma_i^*] \to [\tau_i^*] & \text{and} & \$h_i\ : [\sigma_i^*\ (\texttt{ref null}\ \$ct_i)]\ \text{and} \\ & \$ct_i & : \texttt{cont}\ \$ft_i & \text{and} & \$ft_i : [\tau_i^*] \to [\tau^*]\}_i \end{array}$$

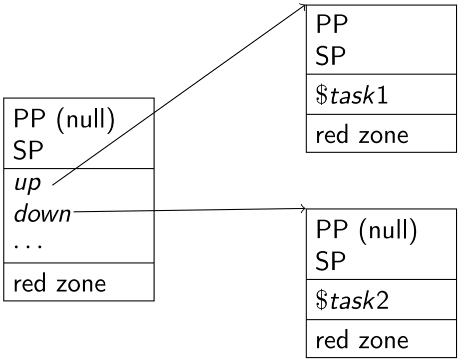
$$\begin{array}{l} \text{and}\ \$ct : \texttt{cont}\ \$ft \\ \text{and}\ \$ft\ : [\sigma^*] \to [\tau^*] \end{array}$$

The instruction fully consume the continuation argument

Refactoring the co2 module (3)

```
(module $co2
  ...                                                      ;; declarations of $task, $yield, etc
  ;; run : [(ref $task) (ref $task)] -> []
  (func $run (export "run") (param $task1 (ref $task)) (param $task2 (ref $task))
    ...                                                    ;; initialisation of $up and $down
    ;; run $up
    (loop $h                                               ;; handling loop
      (block $on_yield (result (ref $ct))
        (resume (tag $yield $on_yield) (local.get $up))    ;; resume $up; handle $yield using $on_yield
        (if (i32.eq (local.get $isOtherDone) (i32.const 1)) ;; $up finished; $down is already done?
          (then (return)))                                 ;; ... then exit
        (local.get $down)                                  ;; ... otherwise prepare to run $down
        (local.set $up)                                    ;; $up := $down
        (local.set $isOtherDone (i32.const 1))             ;; mark other as done
        (br $h)                                            ;; repeat
      )                                                    ;; yield-case definition; stack: [(cont $ct)]
      (local.set $up)                                      ;; set $up to the current continuation
      (if (i32.eqz (local.get $isOtherDone))               ;; is $down already done?
        (then (local.get $down)                            ;; ... then swap $up and $down
              (local.set $down (local.get $up))
              (local.set $up)))
      (br $h)))                                            ;; repeat
)
```

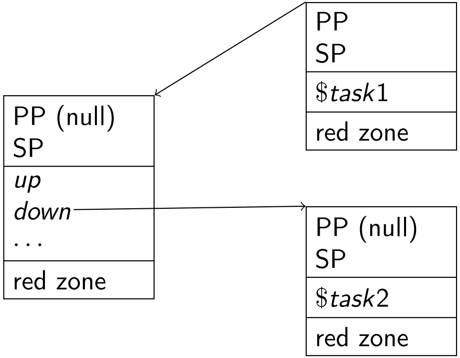
Thinking of **resume** in terms of stacks



resume transfers control from the parent to the child stack

The pointer between parent and child is inverted

Thinking of **resume** in terms of stacks



resume transfers control from the parent to the child stack

The pointer between parent and child is inverted

Instructions: suspending continuations

Continuation suspension

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

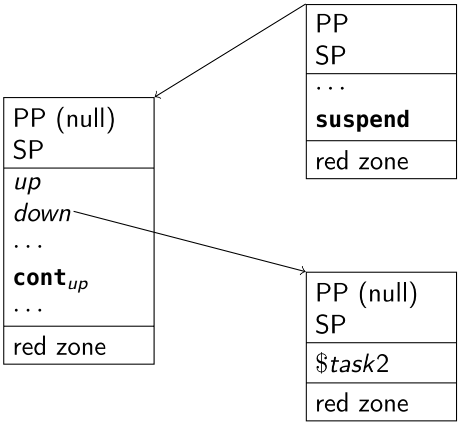
where $\$\mathit{tag} : [\sigma^*] \to [\tau^*]$

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")
    (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`

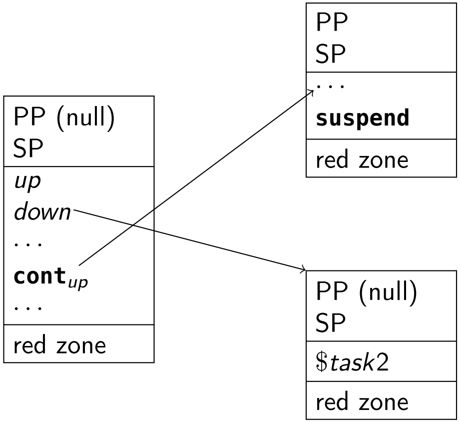
Thinking of **suspend** in terms of stacks



suspend transfers control a child to a (grand)parent

The pointer between child and parent is inverted

Thinking of **suspend** in terms of stacks



suspend transfers control a child to a (grand)parent

The pointer between child and parent is inverted

Current status of the proposal

What has already been done

- Formal specification
- Informal explainer documentation
- Reference implementation

What is happening now

- An implementation in Wasmtime, a production-grade engine

What is going to happen next

- Fine-tune the implementation
- Gathering performance evidence

Summary

Summary

- Effect handlers provide a modular and extensible basis for stack switching in Wasm
- Effect handlers are a proven technology
- The extension to Wasm is minimal and compatible
- Working on a production-grade implementation in Wasmtime

The work is actively being turned into a proposal; for more details see

`https://wasmfx.dev`

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 (2013). "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4.

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

Forster, Yannick et al. (2019). "On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control". In: *J. Funct. Program.* 29, e15.

Hillerström, Daniel (2021). "Foundations for Programming and Implementing Effect Handlers". PhD thesis. The University of Edinburgh, Scotland, UK.

Sivaramakrishnan, K. C. et al. (2021). "Retrofitting effect handlers onto OCaml". In: *PLDI*. ACM, pp. 206–221.

Ghica, Dan et al. (2022). "High-Level Type-Safe Effect Handlers in C++". In: *Proc. ACM Program. Lang.* 6.OOPSLA, pp. 1–30.

Thomson, Patrick et al. (2022). "Fusing industry and academia at GitHub (experience report)". In: *Proc. ACM Program. Lang.* 6.ICFP, pp. 496–511.

Continuation binding, cancellation, and trapping

Partial continuation application

$$\texttt{cont.bind}\ (\texttt{type}\ \$ct) : [\sigma_0^*\ (\texttt{ref null}\ \$ct)] \to [(\texttt{ref}\ \$ct')]$$

$$\begin{aligned} \text{where } \$ct &: \texttt{cont}\ \$ft \text{ and } \$ft : [\sigma_0^*\ \sigma_1^*] \to [\tau^*] \\ \text{and } \$ct' &: \texttt{cont}\ \$ft' \text{ and } \$ft' : [\sigma_1^*] \to [\tau^*] \end{aligned}$$

Continuation cancellation

$$\texttt{resume\_throw}\ (\texttt{tag}\ \$exn)\ (\texttt{tag}\ \$tag\ \$h)^* : [\sigma_0^*\ (\texttt{ref null}\ \$ct)] \to [\tau^*]$$

$$\begin{aligned} \text{where } \ \$exn : [\sigma_0^*] \to [], \{\$tag_i : [\sigma_i^*] \to [\tau_i^*] \ \text{ and } \ \$h_i : [\sigma_i^*\ (\texttt{ref null}\ \$ct_i)] \text{ and} \\ \$ct_i : \texttt{cont}\ \$ft_i \ \text{ and } \ \$ft_i : [\tau_i^*] \to [\tau^*]\}_i \\ \text{and } \ \$ct : \texttt{cont}\ ([\sigma^*] \to [\tau^*] \end{aligned}$$

Control barriers

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

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