WasmFX: Structured Stack Switching for WebAssembly

Daniel Hillerström

Computing Systems Laboratory
Zurich Research Center
Huawei Technologies, Switzerland

September 22, 2023

WebAssembly: a low-level virtual machine (Haas et al. 2017)

What is Wasm?

- A portable bytecode format
- An abstraction of the commonly found hardware
- 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)

Exciting future prospects

- Running non-JavaScript code in the browser
- Secure-by-compilation cloud-native applications
- Efficient cross-platform portable applications

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

- 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

- ~~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

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

Asyncify is the current state-of-the-art (1)

```
(func $doSomething (param $arg i32) (result i32)
  (call $foo
    (call $bar (local.get $arg))))
```

Asyncify is the current state-of-the-art (1)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                    ;; test rewind state
    (then (local.set $arg                                                   ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                              ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                  (then (br $call_bar))                                      ;; restore $call_bar
                  (else (br $restore_foo))))
          (else (br $call_bar))))                                            ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                  (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                  (return (i32.const 0)))  ...))))))
```

Asyncify is the current state-of-the-art (1)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                    ;; test rewind state
    (then (local.set $arg                                                   ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                              ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                  (then (br $call_bar))                                      ;; restore $call_bar
                  (else (br $restore_foo))))
          (else (br $call_bar))))                                            ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                  (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                  (return (i32.const 0)))  ...))))))
```

Asyncify is the current state-of-the-art (1)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                    ;; test rewind state
    (then (local.set $arg                                                   ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                              ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                  (then (br $call_bar))                                      ;; restore $call_bar
                  (else (br $restore_foo))))
          (else (br $call_bar))))                                            ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                  (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                  (return (i32.const 0)))  ...))))))
```

Asyncify is the current state-of-the-art (1)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                    ;; test rewind state
    (then (local.set $arg                                                   ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                              ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                  (then (br $call_bar))                                      ;; restore $call_bar
                  (else (br $restore_foo))))
          (else (br $call_bar))))                                            ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                  (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                  (return (i32.const 0)))  ...))))))
```

Characterising Asyncify

Pros

- Expressive
- Source-to-source transformation
- Optimisable under a closed-world assumption

Cons

- Code size blowup
- Slowdown pure code
- Whole-program approach

Characterising Asyncify

Pros

- Expressive
- Source-to-source transformation
- Optimisable under a closed-world assumption

Cons

- Code size blowup
- Slowdown pure code
- Whole-program approach

But, what is Asyncify? The key primitives are

Unwind stack, delimit unwind, and rewind stack

Characterising Asyncify

Pros

- Expressive
- Source-to-source transformation
- Optimisable under a closed-world assumption

Cons

- Code size blowup
- Slowdown pure code
- Whole-program approach

But, what is Asyncify? The key primitives are

Unwind stack, delimit unwind, and rewind stack

or expressed with a slightly different terminology:

Suspend continuation, delimit suspend, and resume continuation

Characterising Asyncify

Pros

- Expressive
- Source-to-source transformation
- Optimisable under a closed-world assumption

Cons

- Code size blowup
- Slowdown pure code
- Whole-program approach

But, what is Asyncify? The key primitives are

Unwind stack, delimit unwind, and rewind stack

or expressed with a slightly different terminology:

Suspend continuation, delimit suspend, and resume continuation

Asyncify provides a particular implementation of **delimited continuations**!

Characterising Asyncify

Pros

- Expressive
- Source-to-source transformation
- Optimisable under a closed-world assumption

Cons

- Code size blowup
- Slowdown pure code
- Whole-program approach

But, what is Asyncify? The key primitives are

Unwind stack, delimit unwind, and rewind stack

or expressed with a slightly different terminology:

Suspend continuation, delimit suspend, and resume continuation

Asyncify provides a particular implementation of **delimited continuations**!

(the state machine approach dates back at least as far as Adya et al. (2002))

The solution: a delimited continuations instruction set

Main idea

- Let's turn the gist of Asyncify into a proper instruction set!
- . . . but where to start?

Many flavours of delimited continuations

- Felleisen (1988)'s control/prompt
- Danvy and Filinski (1990)'s shift/reset
- Hieb and Dybvig (1990)'s spawn
- Queinnec and Serpette (1991)'s splitter
- Sitaram (1993)'s run/fcontrol
- Gunter, Rémy, and Riecke (1995)'s cupto
- Longley (2009)'s catchcont
- Plotkin and Pretnar (2009)'s effect handlers

(see Appendix A of my PhD thesis (Hillerström 2021) for a comprehensive overview of continuations)

The solution: a delimited continuations instruction set

Main idea

- Let's turn the gist of Asyncify into a proper instruction set!
- . . . but where to start?

Many flavours of delimited continuations

- Felleisen (1988)'s control/prompt
- Danvy and Filinski (1990)'s shift/reset
- Hieb and Dybvig (1990)'s spawn
- Queinnec and Serpette (1991)'s splitter
- Sitaram (1993)'s run/fcontrol
- Gunter, Rémy, and Riecke (1995)'s cupto
- Longley (2009)'s catchcont
- **Plotkin and Pretnar (2009)'s effect handlers**

(see Appendix A of my PhD thesis (Hillerström 2021) for a comprehensive overview of continuations)

Why effect handlers

Design constraints, must work. . .

- . . . without garbage collection
- . . . without closures
- . . . without the use of recursion
- . . . with simply typed stacks
- . . . with imperative control structure
- . . . with predictable cost model
- . . . with legacy code

Why effect handlers

Design constraints, must work. . .

- . . . without garbage collection ✓ reference counting suffices
- . . . without closures
- . . . without the use of recursion
- . . . with simply typed stacks
- . . . with imperative control structure
- . . . with predictable cost model
- . . . with legacy code

Why effect handlers

Design constraints, must work. . .

. . . without garbage collection	✓ reference counting suffices
. . . without closures	✓ first-order abstraction
. . . without the use of recursion	
. . . with simply typed stacks	
. . . with imperative control structure	
. . . with predictable cost model	
. . . with legacy code	

Why effect handlers

Design constraints, must work. . .

- . . . without garbage collection
- . . . without closures
- . . . without the use of recursion
- . . . with simply typed stacks
- . . . with imperative control structure
- . . . with predictable cost model
- . . . with legacy code

✓ reference counting suffices

✓ first-order abstraction

✓ no fundamental dependency

Why effect handlers

Design constraints, must work. . .

. . . without garbage collection	✓ reference counting suffices
. . . without closures	✓ first-order abstraction
. . . without the use of recursion	✓ no fundamental dependency
. . . with simply typed stacks	✓ straightforward typing
. . . with imperative control structure	
. . . with predictable cost model	
. . . with legacy code	

Why effect handlers

Design constraints, must work. . .

. . . without garbage collection	✓ reference counting suffices
. . . without closures	✓ first-order abstraction
. . . without the use of recursion	✓ no fundamental dependency
. . . with simply typed stacks	✓ straightforward typing
. . . with imperative control structure	✓ compatible with native effects
. . . with predictable cost model	
. . . with legacy code	

Why effect handlers

Design constraints, must work...

...without garbage collection	✓ reference counting suffices
...without closures	✓ first-order abstraction
...without the use of recursion	✓ no fundamental dependency
...with simply typed stacks	✓ straightforward typing
...with imperative control structure	✓ compatible with native effects
...with predictable cost model	✓ transparent cost of instructions
...with legacy code	

Why effect handlers

Design constraints, must work. . .

. . . without garbage collection	✓ reference counting suffices
. . . without closures	✓ first-order abstraction
. . . without the use of recursion	✓ no fundamental dependency
. . . with simply typed stacks	✓ straightforward typing
. . . with imperative control structure	✓ compatible with native effects
. . . with predictable cost model	✓ transparent cost of instructions
. . . with legacy code	✓ seamless interop

The WasmFX instruction set extension

Types

- `cont` $\$ft$

Tags

- `tag` $\$tag$ (`param` σ^*) (`result` τ^*)

Core instructions

- `cont.new`
- `resume`
- `suspend`

Other instructions

- `cont.bind`
- `resume_throw`
- `barrier`

We call this extension **WasmFX** (the proposal was originally named “typed continuations”).

`https://wasmfx.dev`

The WasmFX instruction set extension

Types

- `cont` $\$ft$ 🗋 ⚙ ⚙⚙

Tags

- `tag` $\$tag$ $($`param` $\sigma^*)$ $($`result` $\tau^*)$

Core instructions

- `cont.new`
- `resume`
- `suspend`

Other instructions

- `cont.bind`
- `resume_throw`
- `barrier`

Legend

🗋 Spec'ed

⚙ Reference impl.

⚙⚙ Wasmtime impl.

We call this extension **WasmFX** (the proposal was originally named “typed continuations”).

`https://wasmfx.dev`

The WasmFX instruction set extension

Types

- **`cont`** $\$\mathit{ft}$ 🗋 ⚙ ⚙⚙

Tags

- **`tag`** $\$\mathit{tag}$ (**`param`** σ^*) (**`result`** τ^*) 🗋 ⚙ ⚙⚙

Core instructions

- **`cont.new`**
- **`resume`**
- **`suspend`**

Other instructions

- **`cont.bind`**
- **`resume_throw`**
- **`barrier`**

Legend

🗋 Spec'ed

⚙ Reference impl.

⚙⚙ Wasmtime impl.

We call this extension **WasmFX** (the proposal was originally named "typed continuations").

`https://wasmfx.dev`

The WasmFX instruction set extension

Types

- **`cont`** $\$ft$ 🗋 ⚙ ⚙⚙

Tags

- **`tag`** $\$tag$ (**`param`** σ^*) (**`result`** τ^*) 🗋 ⚙ ⚙⚙

Core instructions

- **`cont.new`** 🗋 ⚙ ⚙⚙
- **`resume`**
- **`suspend`**

Other instructions

- **`cont.bind`**
- **`resume_throw`**
- **`barrier`**

Legend

🗋 Spec'ed
⚙ Reference impl.
⚙⚙ Wasmtime impl.

We call this extension **WasmFX** (the proposal was originally named "typed continuations").

`https://wasmfx.dev`

The WasmFX instruction set extension

Types

- **`cont`** $\$\mathit{ft}$ 🗋 ⚙ ⚙⚙

Tags

- **`tag`** $\$\mathit{tag}$ (**`param`** σ^*) (**`result`** τ^*) 🗋 ⚙ ⚙⚙

Core instructions

- **`cont.new`** 🗋 ⚙ ⚙⚙
- **`resume`** 🗋 ⚙ ⚙⚙
- **`suspend`**

Other instructions

- **`cont.bind`**
- **`resume_throw`**
- **`barrier`**

Legend

🗋 Spec'ed

⚙ Reference impl.

⚙⚙ Wasmtime impl.

We call this extension **WasmFX** (the proposal was originally named "typed continuations").

`https://wasmfx.dev`

The WasmFX instruction set extension

Legend

🗋 Spec'ed

⚙ Reference impl.

⚙⚙ Wasmtime impl.

Types

- **`cont`** $\$ft$ 🗋 ⚙ ⚙⚙

Tags

- **`tag`** $\$tag$ $($**`param`** $\sigma^*)$ $($**`result`** $\tau^*)$ 🗋 ⚙ ⚙⚙

Core instructions

- **`cont.new`** 🗋 ⚙ ⚙⚙
- **`resume`** 🗋 ⚙ ⚙⚙
- **`suspend`** 🗋 ⚙ ⚙⚙

Other instructions

- **`cont.bind`**
- **`resume_throw`**
- **`barrier`**

We call this extension **WasmFX** (the proposal was originally named “typed continuations”).

`https://wasmfx.dev`

The WasmFX instruction set extension

Types

- **`cont`** $\$\mathit{ft}$ 🗋 ⚙ ⚙⚙

Tags

- **`tag`** $\$\mathit{tag}$ (**`param`** σ^*) (**`result`** τ^*) 🗋 ⚙ ⚙⚙

Core instructions

- **`cont.new`** 🗋 ⚙ ⚙⚙
- **`resume`** 🗋 ⚙ ⚙⚙
- **`suspend`** 🗋 ⚙ ⚙⚙

Other instructions

- **`cont.bind`** 🗋 ⚙ ⚙⚙
- **`resume_throw`**
- **`barrier`**

Legend

🗋 Spec'ed
⚙ Reference impl.
⚙⚙ Wasmtime impl.

We call this extension **WasmFX** (the proposal was originally named "typed continuations").

`https://wasmfx.dev`

The WasmFX instruction set extension

Types

- **`cont`** $\$\mathit{ft}$ 🗋 ⚙ ⚙⚙

Tags

- **`tag`** $\$\mathit{tag}$ (**`param`** σ^*) (**`result`** τ^*) 🗋 ⚙ ⚙⚙

Core instructions

- **`cont.new`** 🗋 ⚙ ⚙⚙
- **`resume`** 🗋 ⚙ ⚙⚙
- **`suspend`** 🗋 ⚙ ⚙⚙

Other instructions

- **`cont.bind`** 🗋 ⚙ ⚙⚙
- **`resume_throw`** 🗋 ⚙
- **`barrier`**

Legend

🗋 Spec'ed

⚙ Reference impl.

⚙⚙ Wasmtime impl.

We call this extension **WasmFX** (the proposal was originally named "typed continuations").

`https://wasmfx.dev`

The WasmFX instruction set extension

Types

- **`cont`** $\$ft$ 🗋 ⚙ ⚙⚙

Tags

- **`tag`** $\$tag$ (**`param`** σ^*) (**`result`** τ^*) 🗋 ⚙ ⚙⚙

Core instructions

- **`cont.new`** 🗋 ⚙ ⚙⚙
- **`resume`** 🗋 ⚙ ⚙⚙
- **`suspend`** 🗋 ⚙ ⚙⚙

Other instructions

- **`cont.bind`** 🗋 ⚙ ⚙⚙
- **`resume_throw`** 🗋 ⚙
- **`barrier`** 🗋 ⚙

Legend

🗋 Spec'ed
⚙ Reference impl.
⚙⚙ Wasmtime impl.

We call this extension **WasmFX** (the proposal was originally named “typed continuations”).

`https://wasmfx.dev`

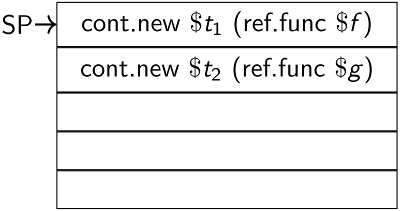
Instruction extension (1)

Continuation allocation

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

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

Operational interpretation of **`cont.new`**

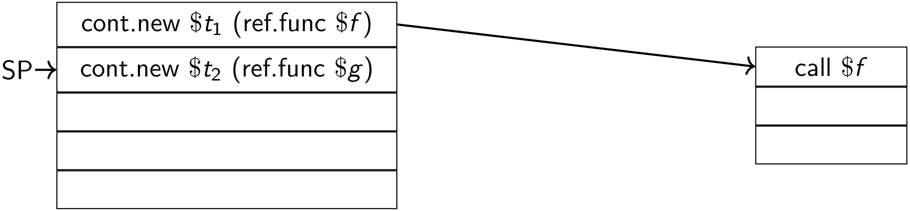


SP→

cont.new $\$t_1$ (ref.func $\$f$)

cont.new $\$t_2$ (ref.func $\$g$)

Operational interpretation of **`cont.new`**

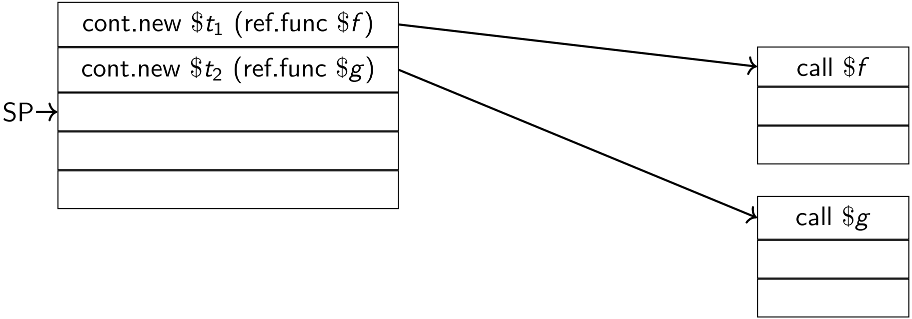


cont.new $\$t_1$ (ref.func $\$f$)

SP→ cont.new $\$t_2$ (ref.func $\$g$)

call $\$f$

Operational interpretation of **`cont.new`**



cont.new $\$t_1$ (ref.func $\$f$)

cont.new $\$t_2$ (ref.func $\$g$)

SP→

call $\$f$

call $\$g$

Instruction extension (2)

Continuation resumption

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

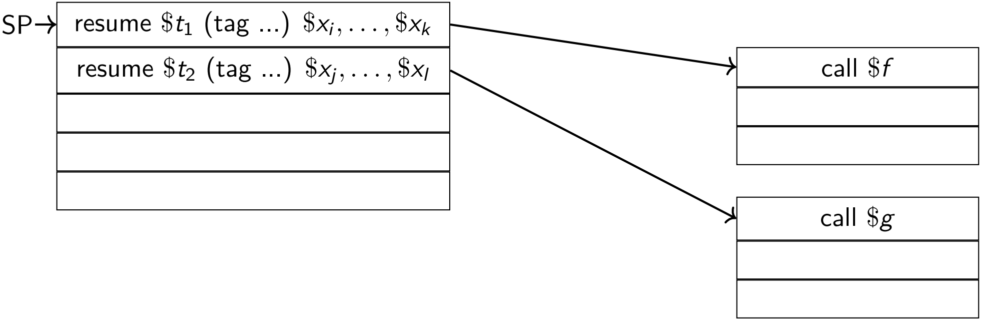
$$\begin{aligned}
\text{where}\quad & \{\$tag_i : [\sigma_i^*] \to [\tau_i^*] \ \text{ and }\ \$h_i : [\sigma_i^*\ (\texttt{\textbf{ref null}}\ \$ct_i)] \text{ and} \\
& \ \ \$ct_i : \texttt{\textbf{cont}}\ \$ft_i \quad\ \text{ and }\ \$ft_i : [\tau_i^*] \to [\tau^*]\}_i \\
& \text{and } \$ct : \texttt{\textbf{cont}}\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

The instruction fully consumes the continuation argument.

Branching from **resume** uses the existing block instructions

```
(func $run (param $task1 (ref $task)) (param $task2 (ref $task))
  (local $up   (ref null $ct))  (local $down (ref null $ct)) ;; locals to manage continuations
  (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)))
  (loop $h                                                   ;; run $up
    (block $on_yield (result (ref $ct))
      (resume (tag $yield $on_yield)
             (local.get $up))
      (if (i32.eq (local.get $isOtherDone) (i32.const 1))    ;; $up finished, check whether $down is done
        (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))                   ;; swap $up and $down
      (then (local.get $down)
            (local.set $down (local.get $up))
            (local.set $up)))
    (br $h)))
```

Operational interpretation of **`resume`**



SP→

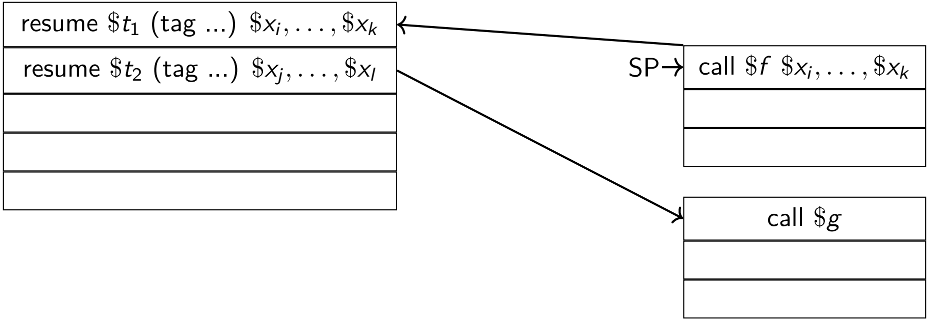
resume $\$t_1$ (tag ...) $\$x_i, \ldots, \$x_k$

resume $\$t_2$ (tag ...) $\$x_j, \ldots, \$x_l$

call $\$f$

call $\$g$

Operational interpretation of **`resume`**



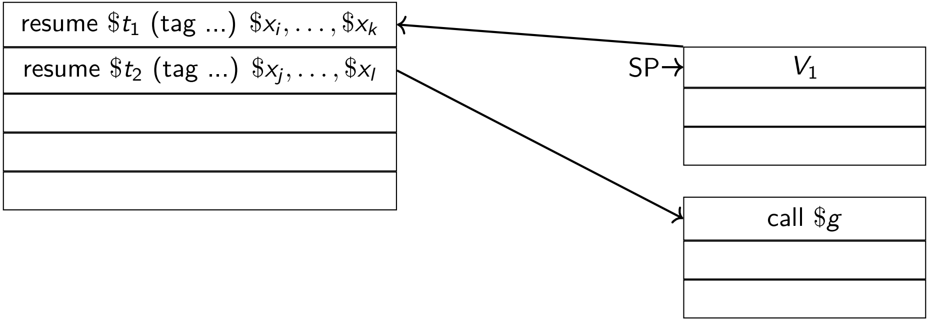
resume $\$t_1$ (tag ...) $\$x_i, \ldots, \$x_k$

resume $\$t_2$ (tag ...) $\$x_j, \ldots, \$x_l$

SP→ call $\$f$ $\$x_i, \ldots, \$x_k$

call $\$g$

Operational interpretation of **resume**



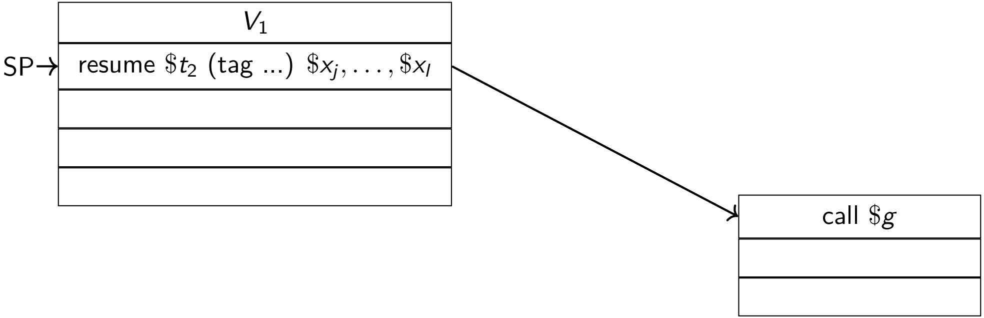
resume $\$t_1$ (tag ...) $\$x_i, \ldots, \$x_k$

resume $\$t_2$ (tag ...) $\$x_j, \ldots, \$x_l$

SP→ V_1

call $\$g$

Operational interpretation of **resume**

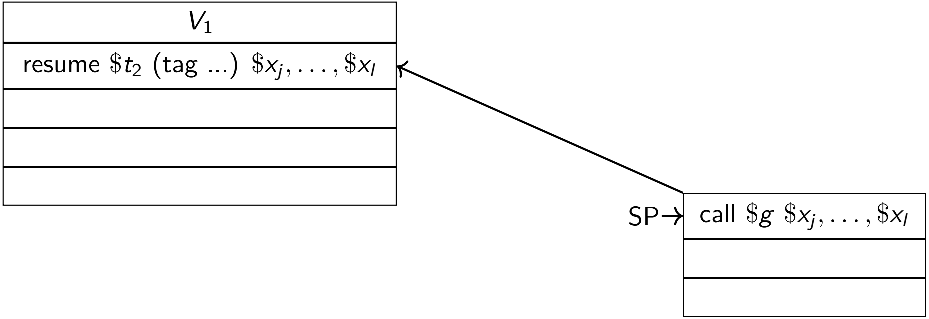


V_1

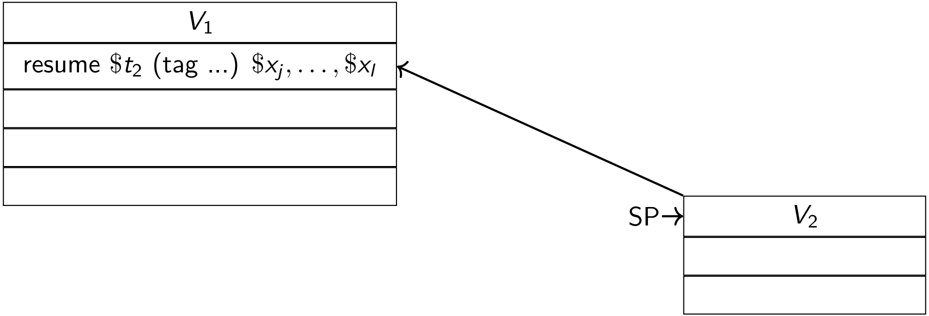
SP→ resume $\$t_2$ (tag ...) $\$x_j, \ldots, \$x_l$

call $\$g$

Operational interpretation of **resume**



Operational interpretation of **resume**

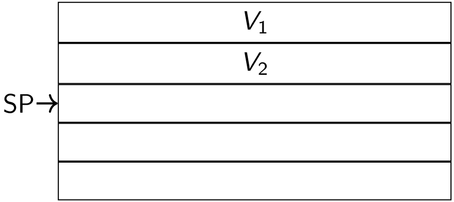


V_1

resume $\$t_2$ (tag ...) $\$x_j, \ldots, \$x_l$

SP→ V_2

Operational interpretation of **`resume`**



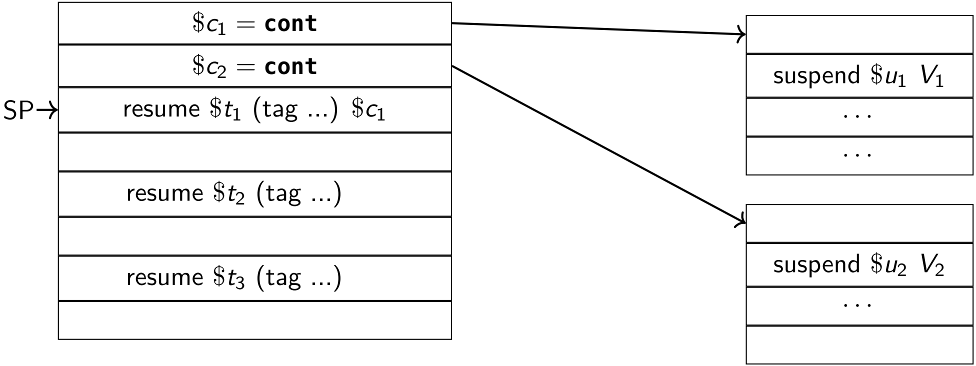
Instruction extension (4)

Continuation suspension

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

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

Operational interpretation of **suspend**



$\$c_1 =$ **cont**

$\$c_2 =$ **cont**

SP→ resume $\$t_1$ (tag ...) $\$c_1$

resume $\$t_2$ (tag ...)

resume $\$t_3$ (tag ...)

suspend $\$u_1$ V_1

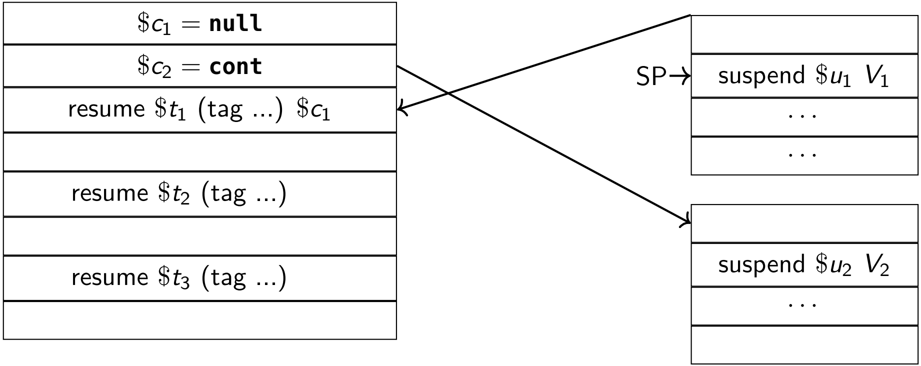
. . .

. . .

suspend $\$u_2$ V_2

. . .

Operational interpretation of **suspend**



$\$c_1 =$ **`null`**

$\$c_2 =$ **`cont`**

resume $\$t_1$ (tag ...) $\$c_1$

resume $\$t_2$ (tag ...)

resume $\$t_3$ (tag ...)

SP→

suspend $\$u_1\ V_1$

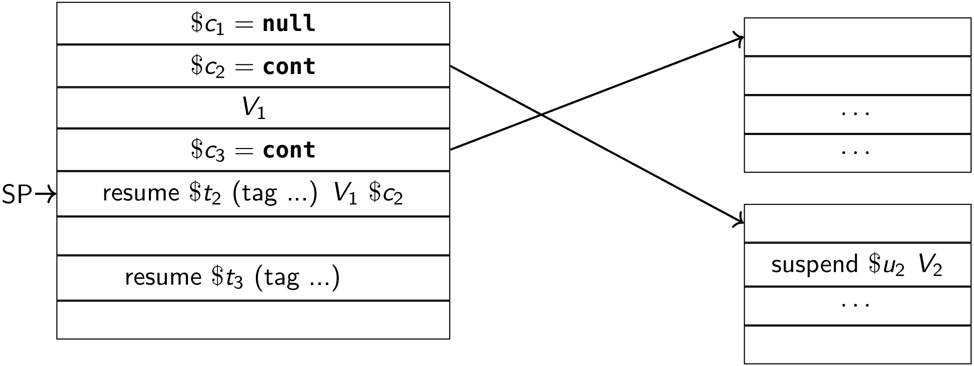
...

...

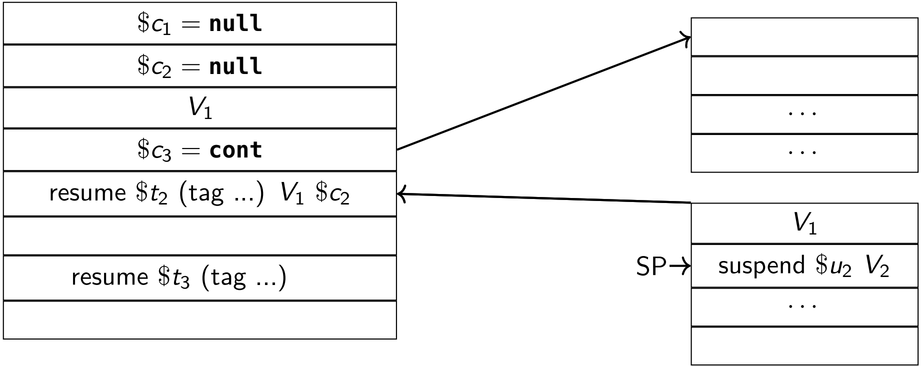
suspend $\$u_2\ V_2$

...

Operational interpretation of **suspend**



Operational interpretation of **suspend**



$\$c_1 = \texttt{null}$

$\$c_2 = \texttt{null}$

V_1

$\$c_3 = \texttt{cont}$

resume $\$t_2$ (tag ...) V_1 $\$c_2$

resume $\$t_3$ (tag ...)

$\cdots$

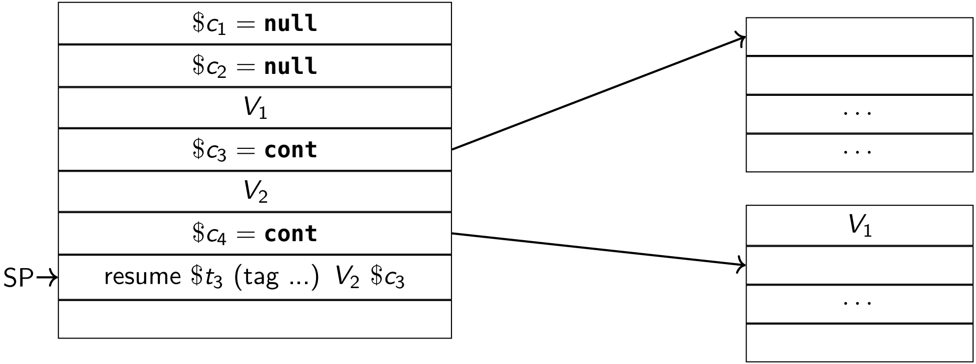
$\cdots$

V_1

SP→ suspend $\$u_2$ V_2

$\cdots$

Operational interpretation of **suspend**



$\$c_1 =$ **`null`**

$\$c_2 =$ **`null`**

V_1

$\$c_3 =$ **`cont`**

V_2

$\$c_4 =$ **`cont`**

SP→ resume $\$t_3$ (tag ...) V_2 $\$c_3$

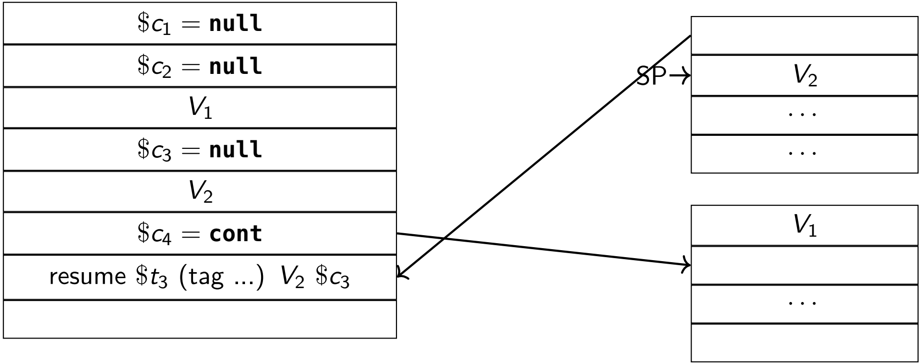
. . .

. . .

V_1

. . .

Operational interpretation of **suspend**



$\$c_1 = \textbf{null}$

$\$c_2 = \textbf{null}$

V_1

$\$c_3 = \textbf{null}$

V_2

$\$c_4 = \textbf{cont}$

resume $\$t_3$ (tag …) V_2 $\$c_3$

SP→

V_2

$\dots$

$\dots$

V_1

$\dots$

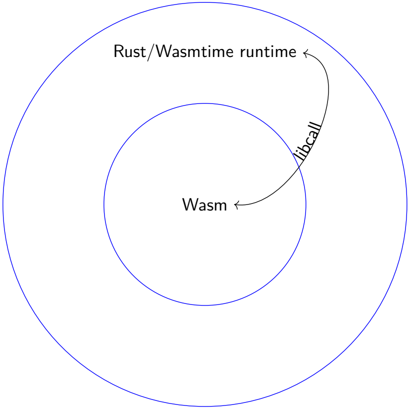
Research prototype implementation in Wasmtime

Prototype implementation in Wasmtime

- Naïve baseline implementation
- Enables running “real” programs
- Stack switching on top of Wasmtime Fiber

Key naïve implementation decisions

- Stack switching is non-Wasm native
- Use `u128` as the universal type
- Reallocate argument buffers on each context switch



Experiments setup

Setup overview

- Fiber-based micro-benchmarks; three implementations: Asyncify, WasmFX, and bespoke
- Fiber interface in C; instantiated with either Asyncify or WasmFX
- No memory leaks allowed

Tools

- WASI SDK version 20.0
- Binaryen version 114

Apples & oranges

- Bespoke and Asyncify implementations are optimised
  - `clang -O3`
  - `wasm-opt -O2 --asyncify --pass-arg=asyncify-ignore-imports`
- WasmFX implementation is unoptimised and assembled by hand
- Different storage
  - Asyncify-backed fibers in linear memory
  - WasmFX-backed fibers in tables
- Tools do not understand function references

Experiments setup: Fiber interface in C

```c
/** The signature of a fiber entry point. **/
typedef void* (*fiber_entry_point_t)(void*);
/** The abstract type of a fiber object. **/
typedef struct fiber* fiber_t;

/** Allocates a new fiber with the default stack size. **/
fiber_t fiber_alloc(fiber_entry_point_t entry);
/** Reclaims the memory occupied by a fiber object. **/
void fiber_free(fiber_t fiber);

/** Yields control to its parent context. This function must be called
    from within a fiber context. **/
void* fiber_yield(void *arg);

/** Possible status codes for ‘fiber_resume‘. **/
typedef enum { FIBER_OK, FIBER_YIELD, FIBER_ERROR } fiber_result_t;

/** Resumes a given ‘fiber‘ with argument ‘arg‘. **/
void* fiber_resume(fiber_t fiber, void *arg, fiber_result_t *result);
```

Microbenchmark: C10m

Description

- HTTP server workload simulation.
- 10 million coroutines in total.
- Sliding window: 10000 coroutines run concurrently, each yielding once.
- Shallow call stack depth

	Run-time ratio	Memory footprint ratio	Binary size ratio
Asyncify	1.00	1.00 (54mb)	1.00 (9.1kb)
WasmFX	0.23	0.98 (55mb)	10.78 (844b)

Microbenchmark: C10m

Description

- HTTP server workload simulation.
- 10 million coroutines in total.
- Sliding window: 10000 coroutines run concurrently, each yielding once.
- Shallow call stack depth

	Run-time ratio	Memory footprint ratio	Binary size ratio
Bespoke	1.00	1.00 (13mb)	1.00 (940b)
Asyncify	0.21	0.24 (54mb)	0.10 (9.1kb)
WasmFX	0.05	0.24 (55mb)	1.11 (844b)

Microbenchmark: Skynet

Description

- Nested tree-structured concurrency simulation.
- 10 million coroutines in total, 6 active, each yielding once.
- No auxiliary data structure; fibers are stored in the control flow state.
- Deep call stack.

	Run-time ratio	Memory footprint ratio	Binary size ratio
Asyncify	1.00	1.00 (14mb)	1.00 (30kb)
WasmFX	2.12	1.00 (14mb)	55.87 (537b)

Microbenchmark: Skynet

Description

- Nested tree-structured concurrency simulation.
- 10 million coroutines in total, 6 active, each yielding once.
- No auxiliary data structure; fibers are stored in the control flow state.
- Deep call stack.

	Run-time ratio	Memory footprint ratio	Binary size ratio
Bespoke	1.00	1.00 (13mb)	1.00 (306b)
Asyncify	0.01	0.01 (14mb)	0.01 (30kb)
WasmFX	0.14	0.24 (14mb)	0.57 (537b)

Future experiments (1)

Benchmarks

- Get into pole position
- Realistic workloads and use-cases

Backends

- Internalise Wasmtime Fiber in codegen
- Cranelift native stack switching

Memory

- Deferred stack allocation
- Stack pools

Extensions

- Named resume blocks
- First-class & generative control tags

Future experiments (2)

Toolchain support

- Compiling control abstractions[1]
- Retrofitting existing toolchains[2]
- Develop new (researchy) toolchains

---

[1] The Kotlin team has shown interest in compiling to the WasmFX instruction set

[2] Currently working on added the WasmFX instruction set to binaryen

WasmFX resource list

Resources

- Formal specification (`https://github.com/wasmfx/specfx/blob/main/proposals/continuations/Overview.md`)
- Informal explainer document (`https://github.com/wasmfx/specfx/blob/main/proposals/continuations/Explainer.md`)
- Reference implementation (`https://github.com/wasmfx/specfx`)
- Research prototype implementation in Wasmtime (`https://github.com/wasmfx/wasmfxtime`)
- Toolchain support (`https://github.com/wasmfx/binaryenfx`)
- OOPSLA'23 research paper (`https://doi.org/10.48550/arXiv.2308.08347`)

`https://wasmfx.dev`

References I

Felleisen, Matthias (1988). “The Theory and Practice of First-Class Prompts”. In: *POPL*. ACM Press, pp. 180–190.

Danvy, Olivier and Andrzej Filinski (1990). “Abstracting Control”. In: *LISP and Functional Programming*, pp. 151–160.

Hieb, Robert and R. Kent Dybvig (1990). “Continuations and Concurrency”. In: *PPoPP*. ACM, pp. 128–136.

Queinnec, Christian and Bernard P. Serpette (1991). “A Dynamic Extent Control Operator for Partial Continuations”. In: *POPL*. ACM Press, pp. 174–184.

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

Gunter, Carl A., Didier Rémy, and Jon G. Riecke (1995). “A Generalization of Exceptions and Control in ML-like Languages”. In: *FPCA*. ACM, pp. 12–23.

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

Adya, Atul et al. (2002). “Cooperative Task Management Without Manual Stack Management”. In: *USENIX Annual Technical Conference, General Track*. USENIX, pp. 289–302.

References II

Longley, John (2009). "Some Programming Languages Suggested by Game Models (Extended Abstract)". In: *MFPS*. Vol. 249. Electronic Notes in Theoretical Computer Science. Elsevier, pp. 117–134.

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.

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

Phipps-Costin, Luna et al. (2023). "Continuing WebAssembly with Effect Handlers". In: *Proc. ACM Program. Lang.* 7.OOPSLA2. To appear.

Instruction extension (3)

Partial continuation application

$$\texttt{cont.bind}\ \$\mathit{sct}\ \$\mathit{dct} : [\sigma_0^*\ (\texttt{ref null}\ \$\mathit{sct})] \to [(\texttt{ref}\ \$\mathit{dct})]$$

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

Instruction extension (5)

Continuation cancellation

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

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

Instruction extension (6)

Control barriers

$$\texttt{\textbf{barrier}}\ \$\mathit{lbl}\ \$\mathit{bt}\ \mathit{instr}^* : [\sigma^*] \rightarrow [\tau^*]$$

where $\$\mathit{bt} = [\sigma^*] \rightarrow [\tau^*]$ and $\mathit{instr}^* : [\sigma^*] \rightarrow [\tau^*]$