Stack Switching in WebAssembly with Effect Handlers

Daniel Hillerström

Computing Systems Laboratory
Zurich Research Center
Huawei Technologies, Switzerland
and
The University of Edinburgh, UK

January 15, 2024

WebAssembly Workshop, London, UK

Collaborators


Sam Lindley


Andreas Rossberg


Daan Leijen


KC Sivaramakrishnan


Matija Pretnar


Frank Emrich


Luna Phipps-Costin


Arjun Guha

`https://wasmfx.dev`

Non-local control is a staple ingredient of many programming languages

OCaml JS ERLANG . . .

- 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)

Non-local control is a staple ingredient of many programming languages

OCaml JS . . .

- 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?

Non-local control is a staple ingredient of many 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 solution

- Add each abstraction as a primitive to Wasm

Non-local control is a staple ingredient of many 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 solution

- ~~Add each abstraction as a primitive to Wasm~~

Non-local control is a staple ingredient of many programming languages

OCaml JS ERLANG . . .

- 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 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 (2)

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

can `$bar` suspend?

Asyncify is the current state-of-the-art (2)

can `$foo` suspend?

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

can `$bar` suspend?

Asyncify is the current state-of-the-art (2)

```
(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 (2)

```
(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 (2)

```
(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 (2)

```
(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
- Obstructs straight-line code
- Whole-program approach

Characterising Asyncify

Pros

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

Cons

- Code size blowup
- Obstructs straight-line 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
- Obstructs straight-line 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
- Obstructs straight-line 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
- Obstructs straight-line 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**!

Intuition: a continuation is a handle to a particular stack

The solution: a delimited continuations instruction set

Main idea

- Let's turn the essence of Asyncify into a bespoke 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 essence of Asyncify into a bespoke 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

Effect handlers provide a structured interface for working with continuations

Andrej Bauer said it best:

$$\begin{array}{rcl} \textbf{effect handlers} & : & \textbf{delimited continuations} \\ & \simeq & \\ \textbf{while} & : & \textbf{goto} \end{array}$$

- Compatible with simple types; synergises with stack typing
- An imperative control structure (like exception handlers)
- Predictable performance
- Works with/without garbage collection (one-shot continuations)

The WasmFX instruction set extension

Types

- **cont** $[\sigma^*] \to [\tau^*]$

Tags

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

Core instructions

- **cont.new**
- **suspend** $\$tag$
- **resume** $(tag\ \$t\ \$h)^*$

We call this instruction set extension **WasmFX**.

The WasmFX instruction set extension

Types

- **cont** $[\sigma^*] \to [\tau^*]$

Tags

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

Core instructions

- **cont.new**
- **suspend** $\$tag$
- **resume** $(tag\ \$t\ \$h)^*$

Other instructions

- **cont.bind**
- **resume_throw** $\$tag$ $(tag\ \$t\ \$h)^*$
- **barrier**

We call this instruction set extension **WasmFX**.

The WasmFX instruction set extension

Types

- **cont** $[\sigma^*] \to [\tau^*]$

Tags

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

Core instructions

- **cont.new**
- **suspend** $\$\mathit{tag}$
- **resume** $(\mathit{tag}\ \$t\ \$h)^*$

Other instructions

- **cont.bind**
- **resume_throw** $\$\mathit{tag}$ $(\mathit{tag}\ \$t\ \$h)^*$
- **barrier**

Legend

Spec'd

Reference impl.

Wasmtime impl.

We call this instruction set extension **WasmFX**.

Example: Yield-style generators

```
(tag $gen (param i32))

(func $nats
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $nats
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sum (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $nats
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sum (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $nats
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sum (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $nats
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sum (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $nats
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sum (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

`(call $sum (i32.const 10) (cont.new (ref.func $nats)))` returns 55

Instructions: creating continuations

Continuation type

$$\texttt{cont}\ [\sigma^*] \to [\tau^*]$$

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

Continuation allocation

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

Instructions: suspending continuations

Continuation suspension

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

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

Instructions: invoking continuations

Continuation resumption

$$\texttt{\textbf{resume}} \quad : [\sigma^* \ (\texttt{\textbf{ref}}\ (\texttt{\textbf{cont}}\ [\sigma^*] \rightarrow [\tau^*]))] \rightarrow [\tau^*]$$

The instruction fully consume the continuation argument

Instructions: invoking continuations

Continuation resumption

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

where $\{\$tag_i : [\sigma_i^*] \to [\tau_i^*]$
$\quad\ \$h_i \quad : [\sigma_i^*\ (\textbf{ref}\ (\textbf{cont}\ [\tau_i^*] \to [\tau^*]))]$

The instruction fully consume the continuation argument

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
  ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))

(func $main-task
  (suspend $spawn (cont.new (ref.func
     $task)))
  (suspend $spawn (cont.new (ref.func
     $task)))
  (suspend $spawn (cont.new (ref.func
     $task)))
  (suspend $spawn (cont.new (ref.func
     $task)))
(func $main
  (call $bfs (cont.new (ref.func
     $main-task))))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Example: lightweight threads

```
(type $taskc (cont [] -> []))
(tag $yield)
   ;; [] -> []
(tag $spawn (param (ref $taskc)))
   ;; [ref $taskc] -> []

(func $task (param $id i32)
  (call $print_i32 (local.get $id))
  (suspend $yield)
  (call $print_i32 (local.get $id)))

(func $main-task
  (suspend $spawn (cont.new (ref.func
     $task)))
  (suspend $spawn (cont.new (ref.func
     $task)))
  (suspend $spawn (cont.new (ref.func
     $task)))
  (suspend $spawn (cont.new (ref.func
     $task)))
(func $main
  (call $bfs (cont.new (ref.func
     $main-task))))
```

```
(func $bfs (param $main (ref $taskc))
  (local $next (ref $taskc))
  (local.set $next (local.get $main))
  (block $on_done
    (loop $schedule_next
      (block $on_spawn (result (ref $taskc) (ref $taskc))
        (block $on_yield (result (ref $taskc))
          (resume (tag $spawn $on_spawn)
                  (tag $yield $on_yield)
                    (local.get $next))
          (br_if $on_done (call $queue-empty))
          (local.set $next (call $dequeue))
          (br $schedule_next)
        ) ;; on_yield
        (call $enqueue)
        (local.set $next (call $dequeue))
        (br $schedule_next)
      ) ;; on_spawn
      (local.set $next)
      (call $enqueue)
      (br $schedule_next)
    ))) ;; on_done
```

Instructions: binding continuations

Partial continuation application

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

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

This instruction fully consumes its continuation argument

Example: lightweight threads (fixed)

```
(type $taskc (cont [] -> []))
(type $itaskc (cont [i32] -> []))

(tag $spawn (param (ref $taskc)))

(func $main-task
  (call $spawn (cont.bind $itaskc $taskc (i32.const 0) (cont.new (ref.func $task))))
  (call $spawn (cont.bind $itaskc $taskc (i32.const 1) (cont.new (ref.func $task))))
  (call $spawn (cont.bind $itaskc $taskc (i32.const 2) (cont.new (ref.func $task))))
  (call $spawn (cont.bind $itaskc $taskc (i32.const 3) (cont.new (ref.func $task)))))
(func $main
  (call $bfs (cont.new $taskc (ref.func $main-task))))
```

Example: lightweight threads (fixed)

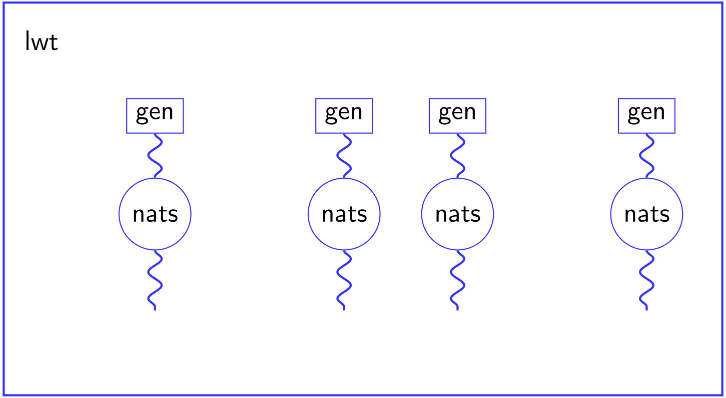
```
(type $taskc (cont [] -> []))
(type $itaskc (cont [i32] -> []))

(tag $spawn (param (ref $taskc)))

(func $main-task
  (call $spawn (cont.bind $itaskc $taskc (i32.const 0) (cont.new (ref.func $task))))
  (call $spawn (cont.bind $itaskc $taskc (i32.const 1) (cont.new (ref.func $task))))
  (call $spawn (cont.bind $itaskc $taskc (i32.const 2) (cont.new (ref.func $task))))
  (call $spawn (cont.bind $itaskc $taskc (i32.const 3) (cont.new (ref.func $task)))))
(func $main
  (call $bfs (cont.new $taskc (ref.func $main-task))))
```

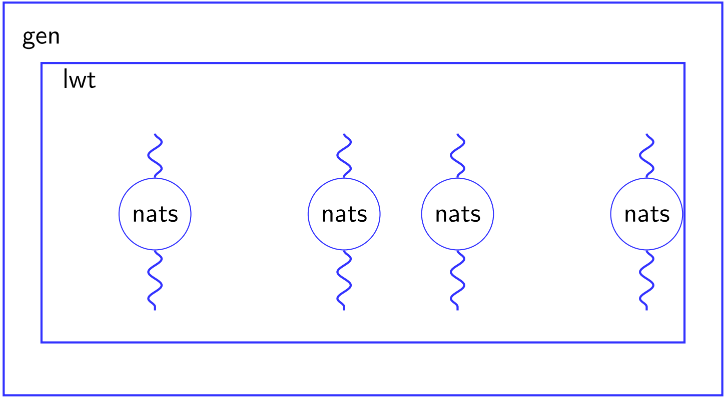
`(call $main)` prints 0 1 2 3 0 1 2 3

Modular composition via effect forwarding (1)



Prints 55 55 55 55

Modular composition via effect forwarding (2)



Prints 190

Instructions: cancelling continuations

Continuation cancellation

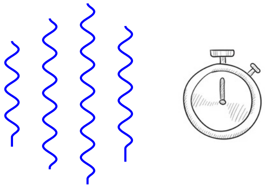
$$\textbf{resume\_throw}\ \$ct\ \$exn\ (\textbf{tag}\ \$tag\ \$h)^* : [\sigma_0^*\ (\textbf{ref}\ \$ct)] \to [\tau^*]$$

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

This instruction fully consumes its continuation argument

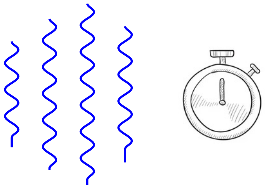
Race to finish with **resume_throw**

```
(tag $cancel) ;; [] -> []
...
(loop $schedule_next
  (block $on_spawn (result (ref $taskc) (ref $taskc))
    (block $on_yield (result (ref $taskc))
      (resume $taskc (tag $spawn $on_spawn)
                     (tag $yield $on_yield) (local.get $next))
       (loop $cleanup
         (br_if $on_done (call $queue-empty))
         (local.set $next (call $dequeue))
         (try
           (do (resume_throw $taskc $cancel
                  (local.get $next)))
           (catch $cancel))
         (br $cleanup)
       ) ;; end of cleanup
...
```

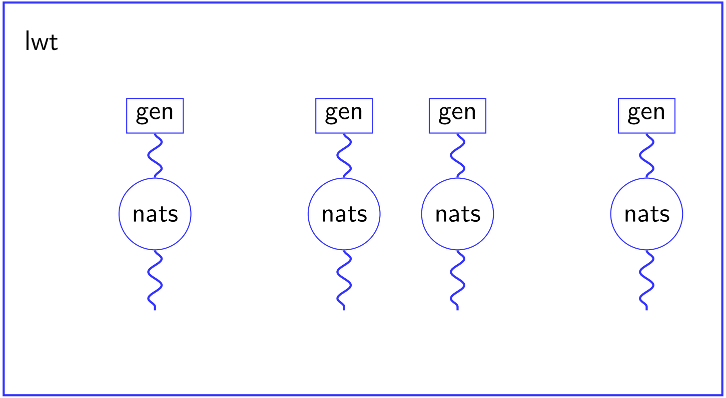


Race to finish with **resume_throw**

```
(tag $cancel) ;; [] -> []
...
(loop $schedule_next
  (block $on_spawn (result (ref $taskc) (ref $taskc))
    (block $on_yield (result (ref $taskc))
      (resume $taskc (tag $spawn $on_spawn)
                     (tag $yield $on_yield) (local.get $next))
       (loop $cleanup
         (br_if $on_done (call $queue-empty))
         (local.set $next (call $dequeue))
         (try
           (do (resume_throw $taskc $cancel
                 (local.get $next)))
           (catch $cancel))
         (br $cleanup)
       ) ;; end of cleanup
...
```



Example: lightweight threads with cancellation



With cancellation prints 55

Characterising the expressive power

Abortive capture, abortive resume (e.g. pthreads)

$$\mathcal{E}[\mathbf{suspend}\ k.M] \rightsquigarrow M[\mathbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\mathbf{resume}\ \mathbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Characterising the expressive power

Abortive capture, abortive resume (e.g. pthreads)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner \mathcal{E} \urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner \mathcal{E}' \urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Abortive capture, composable resume (e.g. effect handlers, shift/reset, etc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner \mathcal{E} \urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner \mathcal{E}' \urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Characterising the expressive power

Abortive capture, abortive resume (e.g. pthreads)

$$\mathcal{E}[\textbf{suspend } k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Abortive capture, composable resume (e.g. effect handlers, shift/reset, etc)

$$\mathcal{E}[\textbf{suspend } k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Composable capture, composable resume (e.g. call/comp-cc)

$$\mathcal{E}[\textbf{suspend } k.M] \rightsquigarrow \mathcal{E}[M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]]$$
$$\mathcal{E}[\textbf{resume cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Characterising the expressive power

Abortive capture, abortive resume (e.g. pthreads)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Abortive capture, composable resume (e.g. effect handlers, shift/reset, etc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Composable capture, composable resume (e.g. call/comp-cc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow \mathcal{E}[M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Composable capture, abortive resume (e.g. call/cc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow \mathcal{E}[M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Characterising the expressive power

Abortive capture, abortive resume (e.g. pthreads)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Abortive capture, composable resume (e.g. effect handlers, shift/reset, etc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Composable capture, composable resume (e.g. call/comp-cc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow \mathcal{E}[M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Composable capture, abortive resume (e.g. call/cc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow \mathcal{E}[M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Characterising the expressive power

Abortive capture, abortive resume (e.g. pthreads)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

Abortive capture, composable resume (e.g. effect handlers, shift/reset, etc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Composable capture, composable resume (e.g. call/comp-cc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow \mathcal{E}[M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}[\mathcal{E}'[M]]$$

Composable capture, abortive resume (e.g. call/cc)

$$\mathcal{E}[\textbf{suspend}\ k.M] \rightsquigarrow \mathcal{E}[M[\textbf{cont}_{\ulcorner\mathcal{E}\urcorner}/k]]$$
$$\mathcal{E}[\textbf{resume}\ \textbf{cont}_{\ulcorner\mathcal{E}'\urcorner}\ M] \rightsquigarrow \mathcal{E}'[M]$$

One-shot continuations can simulate multi-shot semantics (Friedman and Haynes 1985)!

Extensions and variations

Multi-shot continuations

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

Named resume

$$\begin{array}{l}\texttt{resume\_with}\ \$\mathit{hn}\ (\texttt{tag}\ \$\mathit{tag}\ \$h)^* : [\sigma^*\ (\texttt{ref}\ (\texttt{cont}\ (\sigma^*(\texttt{ref handler}\ \tau^*))))] \to [\tau^*]\\ \texttt{suspend\_to}\ \$\mathit{tag} : [\sigma^*(\texttt{ref handler}\ \tau^*)] \to [\tau^*]\end{array}$$

First-class tags

- Dynamic generation of tags
- Pass around tags

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

Friedman, Daniel P. and Christopher T. Haynes (1985). "Constraining Control". In: *POPL*. ACM Press, pp. 245–254.

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.

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.

References II

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

Kammar, Ohad, Sam Lindley, and Nicolas Oury (2013). "Handlers in action". In: *ICFP*. ACM, pp. 145–158.

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, pp. 460–485.