Continuing WebAssembly with Effect Handlers

LUNA PHIPPS-COSTIN, Northeastern University, USA
ANDREAS ROSSBERG, Unaffiliated, Germany
ARJUN GUHA, Northeastern University, USA
DAAN LEIJEN, Microsoft Research, USA
DANIEL HILLERSTRÖM, Huawei Zurich Research Center, Switzerland
KC SIVARAMAKRISHNAN, Tarides and IIT Madras, India
MATIJA PRETNAR, Ljubljana University, Slovenia
SAM LINDLEY, The University of Edinburgh, United Kingdom

WebAssembly (Wasm) is a low-level portable code format offering near native performance. It is intended as a compilation target for a wide variety of source languages. However, Wasm provides no direct support for non-local control flow features such as async/await, generators/iterators, lightweight threads, first-class continuations, etc. This means that compilers for source languages with such features must ceremoniously transform whole source programs in order to target Wasm.

We present *WasmFX*, an extension to Wasm which provides a universal target for non-local control features via *effect handlers*, enabling compilers to translate such features directly into Wasm. Our extension is minimal and only adds three main instructions for creating, suspending, and resuming continuations. Moreover, our primitive instructions are type-safe providing typed continuations which are well-aligned with the design principles of Wasm whose stacks are typed. We present a formal specification of WasmFX and show that the extension is sound. We have implemented WasmFX as an extension to the Wasm reference interpreter and also built a prototype WasmFX extension for Wasmtime, a production-grade Wasm engine, piggybacking on Wasmtime's existing fibers API. The preliminary performance results for our prototype are encouraging, and we outline future plans to realise a native implementation.

Additional Key Words and Phrases: WebAssembly, effect handlers, stack switching

1 INTRODUCTION

WebAssembly (also known as Wasm) [Haas et al. 2017; Rossberg 2019, 2023] is a low-level virtual machine designed to be safe and fast, while being both language- and platform-independent. A primary motivating use case is efficient code execution on the Web, but Wasm is employed in many other environments, such as edge and cloud computing, mobile, embedded systems, and blockchains. Due to its universal nature and its mostly direct mapping to modern CPUs, Wasm is now being targeted by a multitude of different languages.

However, Wasm currently lacks direct support for implementing non-local control flow features appearing in many relevant languages, such as generators/iterators, coroutines, futures/promises, async/await, effect handlers, call/cc, and so forth. For some languages, such features are central to their identity or essential for performance, e.g., to support massively scalable concurrency. Of course, it would be possible to extend Wasm with special support for each individual feature, but that would be at odds with Wasm's low-level spirit and does not scale to the next 700 non-local control flow features. But without native Wasm support, the only option often left to users is a global transformation of source programs, which is at odds with modularity and often inefficient.

Authors' addresses: Luna Phipps-Costin, Northeastern University, USA, phipps-costin.l@northeastern.edu; Andreas Rossberg, Unaffiliated, Germany, rossberg@mpi-sws.org; Arjun Guha, Northeastern University, USA, a.guha@northeastern.edu; Daan Leijen, Microsoft Research, USA, daan@microsoft.com; Daniel Hillerström, Huawei Zurich Research Center, Switzerland, daniel.hillerstrom@ed.ac.uk; KC Sivaramakrishnan, Tarides and IIT Madras, India, kc@kcsrk.info; Matija Pretnar, Ljubljana University, Slovenia, matija.pretnar@fmf.uni-lj.si; Sam Lindley, The University of Edinburgh, United Kingdom, sam.lindley@ed.ac.uk.

We propose WasmFX, a unified structured mechanism that is sufficiently general to cover present use-cases as well as being forwards compatible with future use-cases, while also admitting efficient implementations. WasmFX mechanism is based on *delimited continuations* extended with multiple *named control tags* inspired by Plotkin and Pretnar's effect handlers [Plotkin and Pretnar 2009, 2013]. From an operational perspective, we may view delimited continuations as the rest of a computation from a particular point in its execution up to a *delimiter*. From an implementation perspective, we can view them as additional *stacks* that execution can switch to. Tags, then, are an interface for the possible kinds of non-local transfers of control that a computation may perform. In addition to versatility, effect handlers are supported by a decade's worth of literature [Yallop 2023], have a straightforward typing discipline that aligns well with Wasm, and are proven to admit efficient implementation strategies [Sivaramakrishnan et al. 2021; Xie and Leijen 2021].

Following an overview of Wasm and WasmFX (Section 2), we make the following contributions:

- a formal specification of a minimal extension of Wasm with typed continuations (Section 3),
- a number of applications displaying the versatility of the extended language (Section 4),
- a prototype implementation based on the optimising Wasmtime compiler and a preliminary performance evaluation comparing it to the current current state of the art implementations of non-local control in Wasm (Section 5),
- a discussion of interesting deviations from the usual implementations of typed continuations, implementation concerns that influenced the current design, and potential alternative designs and extensions (Section 6).

We conclude by discussing related and future work.

2 OVERVIEW

2.1 A short Wasm primer

Wasm defines a *virtual instruction set architecture* that closely mirrors the instruction sets common to modern CPUs. Unlike real hardware, however, Wasm is structured as a *stack machine*, i.e., instead of named registers, instructions operate on a virtual operand stack, a design that typically achieves a more compact code representation. For example, a sequence of instructions

```
(i32.const 30) (i32.const 12) (i32.add)
```

pushes two integers to the stack and then adds them, pushing the result. To keep sequences of instructions more readable, the Wasm text format allows syntactic sugar in which operands are "folded" into an instruction. For example, the above sequence can be written as an expression:

```
(i32.add (i32.const 30) (i32.const 12))
```

Local variables. In addition, Wasm also provides *locals*, which are virtual registers accessed via dedicated instructions **local.get** and **local.set**. For example, the of local $i can be incremented:

```
(local.set $i (i32.add (local.get $i) (i32.const 1)))
```

Blocks. In contrast to most other low-level code formats, control flow in Wasm is *structured*: it does not have arbitrary goto, but merely outward branches to the end of a surrounding **block** (like `break` in C) or to the beginning of a **loop** (like `continue` in C). In order to jump out of a loop, we nest the loop immediately inside a block, for example

```
(block $b
  (loop $l
    (br_if $b (i32.ge_u (local.get $i) (i32.const 42)))
    (local.set $i (i32.add (local.get $i) (i32.const 1)))
    (br $l) ) )
```

increments the local $i in a loop. For each iteration, it checks if $i has exceeded 42, in which case it branches to the end of the block $b. Otherwise, it proceeds with incrementing $i and repeats by branching to the beginning of loop $l.

Functions. Wasm code is organised into functions. Besides parameter and result types, functions may declare locals. For example, the function $range processes a given range of sequential integers:

```
(func $range (param $from i32) (param $to i32)  (local $i i32)
  (local.set $i  (local.get $from))
  (block $b
    (loop $l
      (br_if $b  (i32.gt_u  (local.get $i)  (local.get $to)))
      (call $process  (local.get $i))
      (local.set $i  (i32.add  (local.get $i)  (i32.const 1)))
      (br $l) ) ) )
```

It declares a local variable $i, initialising it to the parameter $from and iterating up to $to, processing each value using a previously defined function $process which just prints its argument,

```
(func $process (param $x i32)
  (call $print  (local.get $x))
```

such that the following call prints "10 11 12 13":

```
(call $range  (i32.const 10)  (i32.const 13))
```

Function references. Wasm 2.0 adds simple *reference types*, which represent first-class pointers to functions or host resources [Rossberg 2023]. References are opaque values, such that their actual representation cannot be inspected and their use cannot compromise memory or type safety. The instruction **ref.func** $f produces a reference to a previously defined function $f, whilst **call_ref** calls a function through such a reference operand. If we define $task to be the type of functions that takes a single **i32** parameter, we can write a function $run that calls two such functions sequentially:

```
(type $task (func (param i32)))
(func $run (param $task1 (ref $task)) (param $task2 (ref $task))
  (call_ref $task  (i32.const 10)  (local.get $task1))
  (call_ref $task  (i32.const 20)  (local.get $task2)) )
```

We can define two functions that print given ranges of numbers and pass their references to $run:

```
(func $task1 (param $x i32)  (call $range  (local.get $x)  (i32.const 13)))
(func $task2 (param $x i32)  (call $range  (local.get $x)  (i32.const 23)))
(func $main  (call $run  (ref.func $task1)  (ref.func $task2)))
```

Executing $main prints "10 11 12 13 20 21 22 23".

2.2 Continuations

Let us now turn from plain Wasm to WasmFX. A *continuation* is a first-class program object representing the remaining computation from a certain point in the execution of a program. WasmFX is based on a structured notion of a *delimited continuation*, which is a continuation whose extent is delimited by some *control delimiter*, meaning it represents the remaining computation from a certain point up to (and possibly including) its delimiter. Intuitively, a continuation represents the current stack, whereas a delimited continuation represents a segment of the stack. In implementations, stack segments are often realised by allocating *multiple* stacks in a dynamic parent-child hierarchy.

WasmFX introduces three core instructions for manipulating delimited continuations. First, **cont .new** creates a new continuation from a given function. At the implementation level, this can be thought of as *creating a new stack*. Next, **suspend** suspends the currently running computation and reifies it into a continuation. In terms of implementation, this can be viewed as *switching* from the current stack to its parent. Finally, **resume** invokes a given continuation in a delimited scope and declares how to handle any suspensions that happen inside it. In an implementation, that means switching from a parent stack to a selected child. For example, the code

```
(type $cont (cont $task))
(cont.new $cont  (ref.func $task1))
```

declares the type $cont to be continuations of the function type $task, and creates a continuation that represents a suspended computation to print a sequence of integers.

2.3 Tags

A running computation suspends itself by invoking a declared *control tag*, which transfers control to the nearest *handler* for that tag (Section 2.4). This way, control tags are similar to exceptions, with the key difference being that execution may subsequently be resumed by the handling context. Tags, however, are not only resumable exceptions [Steele Jr. 1990], as handlers not only return a result to the suspended computation, but can control when and how it is resumed.

For example, let us declare a tag $yield that allows a current computation to signal (e.g. to a scheduler) that it is ready to relinquish control of execution. We can then extend the function $process to yield after printing out the given number:

```
(tag $yield)
(func $process (param $x i32)
  (call $print  (local.get $x))
  (suspend $yield) )
```

However, **suspend** only creates a continuation at the invocation site of the control tag, it does not determine how to handle it (in contrast to delimited continuations, which do both). Thus, executing

```
(call $run  (ref.func $task2)  (ref.func $task1))
```

prints 0 and then traps, as there is no handler associated with $yield.

2.4 Handlers

How to react to $yield is specified by installing a suitable *handler*, which may be done each time a continuation is resumed. A handler both delimits the scope of continuations and determines the behaviour of subsequent control suspensions inside them. The **resume** instruction defines a handler in terms of a jump table pairing control tags with labels pointing to corresponding handler code.

Consider the following example, which treats $yield as a non-resumable exception. We begin by setting a local variable $k to store a reference to the continuation.

```
(local $k (ref $cont))
(cont.new $cont  (ref.func $task1))
(local.set $k)
```

Next, we start an encompassing block $h, whose label serves as a join point. Immediately inside $h, we define another block with label $on_yield, which will serve as a label for the handling code. Recall that in Wasm, branching to a block continues execution *after* the block, thus the code used to handle $yield is that immediately after the $on_yield block. Handler code receives the suspended continuation on the stack, thus the block is annotated with a **ref** $cont result type.

```
(block $h
  (block $on_yield (result (ref $cont))
    (resume $cont (on $yield $on_yield)  (local.get $k))
    (call $print  (i32.const −2))
    (br $h) )
  ;; $on_yield lands here
  (call $print  (i32.const −1))
  (br $h) )
```

Having set both blocks up, we may resume the continuation using **resume**, which points to the handling code for $yield and invokes the continuation referenced by $k. The code that follows the resumption describes printing -2, but will never execute since the continuation will suspend with $yield. Thus the program prints -1 and exits. The ultimate output is the number 10, printed by $task1, followed by -1, printed by the handling code.

The **resume** instruction pushes the continuation created by (**suspend** $yield) to the stack before branching to a handler. For example, we may store it back into $k and repeat the process by introducing a loop $l.

```
(block $h
  (loop $l
    (block $on_yield (result (ref $cont))
      (resume $cont (on $yield $on_yield)  (local.get $k))
      (call $print  (i32.const −2))
      (br $h) )
    ;; $on_yield lands here, with the continuation on the stack
    (local.set $k)  ;; grab continuation and save it
    (call $print  (i32.const −1))
    (br $l) ) )
```

This then alternates between printing numbers from $task1 and printing -1, ultimately printing -2 once the continuation runs to its end. In terms of implementation, this can be thought of as switching back and forth between the main stack and the stack for $task1.

The **resume** instruction consumes its continuation operand, meaning a continuation may be resumed only once — i.e., we only support *single-shot* continuations. An attempt to resume the same continuation again will result in a trap (an extension to support multi-shot continuations would be interesting, but difficult to support efficiently or robustly in existing Wasm engines). When a continuation runs to completion, i.e., when control is transferred back to **resume** via a return rather than a suspension, then no further continuation is created. The underlying child stack can be considered dead and be reclaimed by the system.

2.5 Using continuations for scheduling

Handlers give us fine control over how to resume continuations. For example, we can use them to implement lightweight threads (a primary use-case for handlers). Let us assume a suitable interface for a queue implementation in which we keep inactive threads. While the queue is non-empty, the scheduler repeatedly resumes the thread at the head of the queue, by way of a loop $l containing a block $on_yield in which the continuation is resumed. On any $yield suspension, we branch to the instructions following the block, which enqueue the current continuation and repeat the process:

```
(block $scheduler
 (loop $l
  (br_if $scheduler (call $queue_empty)) ;; exit if the queue is empty
  (block $on_yield (result (ref $cont))
   (resume $cont (on $yield $on_yield)  (call $dequeue))
   ;; we land here when a thread finishes without suspensions
   (br $l) ) ;; repeat the loop and proceed with a smaller queue
  ;; we land here when a thread suspends with $yield, with continuation on stack
  (call $enqueue)
  (br $l) ) )
```

Now, enqueuing tasks $task1 and $task2 as before and running the scheduler, we get interleaved output: "10 20 11 21 12 22 23 33".

3 LANGUAGE

3.1 Syntax

In the formal definition in this paper, we focus on a minimal language *WasmFX*, given in Figure 1, and omit much of Wasm's complete feature set [Rossberg 2023], since it is largely orthogonal to our proposal (features omitted relative to Wasm 2.0 include additional control instructions, arithmetics, floating point and vector types, subtyping, globals, tables, and linear memory). Nevertheless, the proposal is compatible with the full language, and in examples we take the liberty to assume a more complete instruction set, e.g., obvious features such as arithmetics.

Our proposal builds on top of extensions of Wasm with function references [Rossberg 2022], which we extend to continuations, and with exceptions [Ahn 2022], from which we adapt *tag* declarations. We also use exceptions whenever we want to abort a continuation that we do not intend to resume. Both of these extensions are not yet in the standard, but part of proposals that are close to reaching the final states of standardisation. Features added by our proposal are highlighted in grey in our figures. But let us turn to the pre-existing constructs (in black) first.

Types. As a low-level language, Wasm only provides primitive data types available in hardware, such as integer numbers of different bit width, such as i32 and i64. To support function references, Wasm 2.0 introduced *reference types* of the form ref ft, where a *function type* is of the form $t_1^* \to t_2^*$,

(index, label)	x, l	$\in$	$\mathbb{N}$
(value type)	t	$::=$	i32 | i64 | ref ft
(function type)	ft	$::=$	$\mathrm{cont}^?\ t^* \to t^*$
(instruction)	e	$::=$	t.**const** i | **local.get** x | **local.set** x | **block**$_{ft}$ e^* | **loop**$_{ft}$ e^* | **br** l | **br_if** l | **call_ref**$_{ft}$ | **ref.null**$_{ft}$ | **ref.func** x | **throw** x | **cont.new**$_{ft}$ | **resume**$_{ft}$ h^* | **suspend** x | **cont.bind**$_{ft,ft}$ | **resume_throw**$_{ft}$ x h^* |
(handler clause)	h	$::=$	**on** x l
(function)	f	$::=$	**func** ft t^* e^*
(tag)	tg	$::=$	**tag** ft
(module)	m	$::=$	**module** tg^* f^*

Fig. 1. WasmFX Syntax

mapping arguments t_1^* to results t_2^*. In our extension, continuations are assigned a similar type, except annotated with a cont keyword.

Instructions. The core building block of Wasm are *instructions*. The instruction set we consider here includes basic constants, locals, control blocks **block** and **loop**, and basic branches **br** and **br_if**. Certain instructions are annotated with types (type-set as subscripts) to ensure unique typing. In examples, we denote locals, functions, tags, or labels through symbolic names of the form $name, but these just stand in for integer de Bruijn indices that index into respective definition lists.

We include **call_ref** [Rossberg 2022], which performs a function call through a first-class function, a null reference **ref.null**, and a function reference **ref.func**. When calling a reference, a runtime check that it is not null occurs; if it is, execution *traps*, i.e. immediately aborts. Given **call_ref**, a regular **call** x instruction invoking function x, as available in bare Wasm, can be viewed as a shorthand for the instruction sequence (**ref.func** x) ($\textbf{call\_ref}_{ft}$), so we omit it from our core. After declaring an exception tag x, we may throw it using the **throw** x instruction. The exception proposal [Ahn 2022] also contains an instruction for handling exceptions, which we omit for simplicity.

Next, the new instructions! As described in Section 2, $\textbf{cont.new}_{ft}$ creates a new continuation (i.e., a stack) from a function reference, $\textbf{resume}_{ft}\ h^*$ resumes it under the specified handlers h^*, and **suspend** x suspends the current continuation and transfers control to the handler for tag x.

WasmFX adds two more instructions for special purposes. The instruction **cont.bind** partially applies a given continuation. Unlike partial application for function references, this instruction requires no allocation: since a continuation can be resumed only once, an implementation can modify it in-place. To abort and finalise a continuation, we reuse the existing exception mechanism: the **resume_throw** $x\ h^*$ instruction injects an exception x at the suspension site and thereby unwinds the aborted stack. Correct code is expected to consume all continuations *linearly*, i.e., either resume them or abort them explicitly with **resume_throw** — both will ultimately terminate the computation and reclaim the stack. This prevents memory leaks and triggers any user defined resource finalisation that was in place when the continuation was suspended (e.g., through exception handlers). Handling exception x may trigger further suspensions, which are handled with h^*.

Modules. All Wasm code is organised into modules. Each module encapsulates local definitions, such as functions or tags, which can optionally be exported or imported from other modules. Here, we only include the bare minimum of concepts from modules. In particular, we assume a single global module and omit the ability to export or import definitions, which is needed to link multiple modules together. In particular, the full language allows importing/exporting tags.

A tag definition includes a function type whose argument types determine the arguments of the exception. For an exception, the result of this function type must be empty, but when used as control tags, the result types specify values to be transferred back upon resumption.

3.2 Typing

Wasm code is typed in a context C recording types of locals, functions, tags and labels. Recall that Wasm uses de Bruijn indices, thus contexts are simply indexed lists for each namespace. We denote the type associated with label l by $C_{\mathsf{label}}(l)$, and similarly for other constructs.

$$\text{(contexts)}\ \ C \ ::= \ \epsilon \mid C, \mathsf{func}\ ft \mid C, \mathsf{tag}\ ft \mid C, \mathsf{local}\ t \mid C, \mathsf{label}\ t^*$$

To each Wasm instruction we assign a function type through a judgement $C \vdash e : t_1^* \to t_2^*$ describing the types t_1^* of values it expects to pop off the stack and the types t_2^* of values it is expected to push back. The typing rules for instructions are given in Figure 2.

$$\boxed{C \vdash e^* : t_1^* \to t_2^*} \qquad \boxed{C \vdash h : t^* \text{ clause}}$$

$$\frac{}{C \vdash \epsilon : \epsilon \to \epsilon} \qquad \frac{C \vdash e_1 : t_1^* \to t_2^* \qquad C \vdash e_2^* : t_2^* \to t_3^*}{C \vdash e_1\ e_2^* : t_1^* \to t_3^*} \qquad \frac{C \vdash e^* : t_1^* \to t_2^*}{C \vdash e^* : t\ t_1^* \to t\ t_2^*}$$

$$\frac{}{C \vdash t.\mathbf{const}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{local}}(x) = t}{C \vdash \mathbf{local.get}\ x : \epsilon \to t} \qquad \frac{C_{\mathsf{local}}(x) = t}{C \vdash \mathbf{local.set}\ x : t \to \epsilon}$$

$$\frac{ft = t_1^* \to t_2^* \qquad C, \mathsf{label}\ (t_2^*) \vdash e^* : ft}{C \vdash \mathbf{block}_{ft}\ e^* : ft} \qquad \frac{ft = t_1^* \to t_2^* \qquad C, \mathsf{label}\ (t_1^*) \vdash e^* : ft}{C \vdash \mathbf{loop}_{ft}\ e^* : ft}$$

$$\frac{C_{\mathsf{label}}(l) = t^*}{C \vdash \mathbf{br}\ l : t_1^*\ t^* \to t_2^*} \qquad \frac{C_{\mathsf{label}}(l) = t^*}{C \vdash \mathbf{br\_if}\ l : t^*\ \mathsf{i32} \to t^*} \qquad \frac{ft = t_1^* \to t_2^*}{C \vdash \mathbf{call\_ref}_{ft} : t_1^*\ (\mathsf{ref}\ ft) \to t_2^*}$$

$$\frac{}{C \vdash \mathbf{ref.null}_{ft} : \epsilon \to (\mathsf{ref}\ ft)} \qquad \frac{C_{\mathsf{func}}(x) = ft}{C \vdash \mathbf{ref.func}\ x : \epsilon \to (\mathsf{ref}\ ft)} \qquad \frac{C_{\mathsf{tag}}(x) = t^* \to \epsilon}{C \vdash \mathbf{throw}\ x : t_1^*\ t^* \to t_2^*}$$

$$\frac{}{C \vdash \mathbf{cont.new}_{ft} : (\mathsf{ref}\ ft) \to (\mathsf{ref\ cont}\, ft)} \qquad \frac{ft = t_1^* \to t_2^* \qquad (C \vdash h : t_2^* \text{ clause})^*}{C \vdash \mathbf{resume}_{ft}\ h^* : t_1^*\ (\mathsf{ref\ cont}\, ft) \to t_2^*}$$

$$\frac{C_{\mathsf{tag}}(x) = t_1^* \to t_2^*}{C \vdash \mathbf{suspend}\ x : t_1^* \to t_2^*} \qquad \frac{ft = \mathsf{cont}\ t^*\ t_1^* \to t_2^* \qquad ft' = \mathsf{cont}\ t_1^* \to t_2^*}{C \vdash \mathbf{cont.bind}_{ft,ft'} : t^*\ (\mathsf{ref}\ ft) \to (\mathsf{ref}\ ft')}$$

$$\frac{ft = t_1^* \to t_2^* \qquad C_{\mathsf{tag}}(x) = t^* \to \epsilon \qquad (C \vdash h : t_2^* \text{ clause})^*}{C \vdash \mathbf{resume\_throw}_{ft}\ x\ h^* : t^*\ (\mathsf{ref\ cont}\, ft) \to t_2^*}$$

$$\frac{C_{\mathsf{tag}}(x) = t_1'^* \to t_2'^* \qquad C_{\mathsf{label}}(l) = t_1'^*\ (\mathsf{ref\ cont}\ (t_2'^* \to t_2^*))}{C \vdash \mathbf{on}\ x\ l : t_2^* \text{ clause}}$$

$$\boxed{C \vdash tg : t_1^* \to t_2^*} \qquad \boxed{C \vdash f : t_1^* \to t_2^*} \qquad \boxed{\vdash m \text{ ok}}$$

$$\frac{ft = t_1^* \to t_2^*}{C \vdash \mathbf{tag}\ ft : ft} \qquad \frac{ft = t_1^* \to t_2^* \qquad C, \mathsf{local}\ t_1^*\ t^* \vdash e^* : \epsilon \to t_2^*}{C \vdash \mathbf{func}\ ft\ t^*\ e^* : ft}$$

$$\frac{C = \mathsf{func}\ ft^*, \mathsf{tag}\ ft'^* \qquad (C \vdash tg : ft')^* \qquad (C \vdash f : ft)^*}{\vdash \mathbf{module}\ tg^*\ f^* \text{ok}}$$

Fig. 2. Typing rules for WasmFX

The first half of the figure contains the typing rules for plain Wasm instructions, essentially unchanged from Haas et al. [2017]. We add the rules for **call_ref**, **ref.null**, **ref.func** and **throw** taken from aforementioned proposals, which should be self-explanatory.[1]

[1]The function references proposal [Rossberg 2022] distinguishes non-nullable references, but we omit that distinction.

$$
\begin{array}{lrcl}
\text{(cont address)} & k & \in & \mathbb{N} \\
\text{(admin instruction)} & e & ::= & \dots \mid \mathbf{trap} \mid \mathbf{ref.cont}\ k \mid \\
 & & & \mathbf{label}_n\{e^*\}\ e^* \mid \mathbf{frame}\{\mathit{ff}\}\ e^* \mid \mathbf{handler}\{h^*\}\ e^* \\
\text{(value)} & v & ::= & t.\mathbf{const}\ i \mid \mathbf{ref.null}_{ft} \mid \mathbf{ref.func}\ x \mid \mathbf{ref.cont}\ k \\
\text{(evaluation context)} & E & ::= & [\_] \mid v^*\ E\ e^* \mid \mathbf{label}_n\{e^*\}\ E \mid \mathbf{handler}\{h^*\}\ E \\
\text{(handler context)} & E^x & ::= & [\_] \mid v^*\ E^x e^* \mid \mathbf{label}_n\{e^*\}\ E^x \mid \mathbf{frame}\{\mathit{ff}\}\ E^x \mid \mathbf{handler}\{h^*\}\ E^x\ \ (x \notin h^*) \\
\\
\text{(continuation)} & \mathit{cont} & ::= & E^x \mid \dagger \\
\text{(store)} & s & ::= & \{\mathrm{func}\ f^*, \mathrm{tag}\ tg^*, \mathrm{cont}\ \mathit{cont}^*\} \\
\text{(frame)} & \mathit{ff} & ::= & \{\mathrm{local}\ v^*\} \\
\text{(configuration)} & c & ::= & s; \mathit{ff}; e^*
\end{array}
$$

Fig. 3. Syntax of WasmFX reductions

Note that **br** and **throw** are *stack-polymorphic*: they allow arbitrary inputs t_1^* and outputs t_2^* to be assumed for the stack — that is fine, because these instructions pass control unconditionally and never return.

Let's turn our attention to what's new. The **cont.new** instruction converts a function reference into a continuation reference of analogous type. The **resume** instruction expects a continuation of type $t_1^* \rightarrow t_2^*$ on the stack. The stack must also hold values of types t_1^* to be consumed by the continuation. If the continuation terminates and hence **resume** returns, it yields results of type t_2^*. The handler clauses are assigned types matching the result type of the continuation (t_2^*). The auxiliary judgement $C \vdash \mathbf{on}\ x\ l : t_2^*$ clause defined at the end ensures that each label l used to handle a tag x expects the corresponding tag arguments $t_1'^*$ and a continuation reference as parameters. The continuation will produce t_2^* (corresponding to the type of the handler) once supplied with the tag results $t_2'^*$. Typing of **resume_throw** is analogous, except that it expects arguments for the thrown tag x instead of the continuation parameters.

The **suspend** instruction behaves similarly to **throw** in that it expects appropriate tag operands of type t_1^*. The main difference is that the continuation can be subsequently resumed with values of type t_2^*. The **cont.bind** instruction consumes and produces a continuation reference, which is the same except that its leading arguments t^* have been partially applied.

Typing rules for the included module-level constructs are straightforward. When we declare a function of a type $t_1^* \rightarrow t_2^*$, we may declare locals of type t^* in addition to the locals of type t_1^* that hold the function's arguments. The context C for typing a module is constructed recursively from the types of the individual definitions. In effect, a module is one big recursive definition.

3.3 Execution

The operational semantics of Wasm is given in terms of small-step reductions between configurations consisting of the executed instruction sequence together with a global store and the current stack frame. To concisely express the reductions, we introduce additional syntax in Figure 3. Again, it is largely inherited from plain Wasm [Haas et al. 2017; Rossberg 2023]; we focus on the novelties.

Instructions are extended into *administrative instructions*, which represent internal values or operators not directly expressible in Wasm's source instruction set. The new instruction **ref.cont** k, which also is a value, represents a reference to a continuation allocated at address k in the global store. The instruction $\mathbf{handler}\{h^*\}\ e^*$ represents execution of e^* under an active handler h^*. Handlers can occur as part of evaluation contexts. For specifying the semantics of suspension, we use a restricted version of *handler contexts* E^x, which are similar to evaluation contexts, except

$$
\begin{array}{llll}
\mathit{ff}[x=v];(\mathbf{local.get}\,x) & \hookrightarrow & \mathit{ff}[x=v];v & \\
\mathit{ff}[x=v];v'\,(\mathbf{local.set}\,x) & \hookrightarrow & \mathit{ff}[x=v'];\epsilon & \\
v^n\,(\mathbf{block}_{ft}\ e^*) & \hookrightarrow & (\mathbf{label}_m\{\epsilon\}\,v^n\,e^*) & \text{if } ft = t_1^n \to t_2^m \\
v^n\,(\mathbf{loop}_{ft}\ e^*) & \hookrightarrow & (\mathbf{label}_n\{(\mathbf{loop}_{ft}\ e^*)\}\,v^n\,e^*) & \text{if } ft = t_1^n \to t_2^m \\
(\mathbf{label}_n\{e^*\}\,v^*) & \hookrightarrow & v^* & \\
(\mathbf{label}_n\{e_1^*\}\,v^*\,v^n\,(\mathbf{br}\,0)\,e_2^*) & \hookrightarrow & v^n\,e_1^* & \\
(\mathbf{label}_n\{e_1^*\}\,v^*\,(\mathbf{br}\,l{+}1)\,e_2^*) & \hookrightarrow & v^*\,(\mathbf{br}\,l) & \\
(\mathbf{handler}\{h^*\}\,v^*\,(\mathbf{br}\,l)\,e_2^*) & \hookrightarrow & v^*\,(\mathbf{br}\,l) & \\
(\text{i32}.\mathbf{const}\,i)\,(\mathbf{br\_if}\,l) & \hookrightarrow & (\mathbf{br}\,l) & \text{if } i \neq 0 \\
(\text{i32}.\mathbf{const}\,i)\,(\mathbf{br\_if}\,l) & \hookrightarrow & \epsilon & \text{if } i = 0 \\
s;v^n\,(\mathbf{ref.func}\ x)\,(\mathbf{call\_ref}_{ft}) & \hookrightarrow & s;(\mathbf{frame}\{\text{local}\,v^n\,(\text{i32}.\mathbf{const}\,0)^k\}\,e^*) & \\
 & & \text{if } s_{\text{func}}[x] = \text{func}\ (t_1^n \to t_2^m)\ t^k\ e^* & \\
(\mathbf{frame}\{\mathit{ff}\}\,E[e^*]) & \hookrightarrow & (\mathbf{frame}\{\mathit{ff}'\}\,E[e'^*]) & \\
 & & \text{if } \mathit{ff};e^* \hookrightarrow \mathit{ff}';e'^* & \\
(\mathbf{frame}\{\mathit{ff}\}\,v^*) & \hookrightarrow & v^* & \\
s;(\mathbf{frame}\{\mathit{ff}\}\,E[v^n\,(\mathbf{throw}\,x)]) & \hookrightarrow & s;v^n\,(\mathbf{throw}\,x) & \\
 & & \text{if } s_{\text{tag}}[x] = t^n \to t^* & \\
s;(\mathbf{ref.func}\ x)\,(\mathbf{cont.new}_{ft}) & \hookrightarrow & s[+k=E];(\mathbf{ref.cont}\ k) & \\
 & & \text{where } E = [\_]\ (\mathbf{ref.func}\ x)\ (\mathbf{call\_ref}_{ft}) & \\
s[k=E];v^n\,(\mathbf{ref.cont}\ k)\,(\mathbf{resume}_{ft}\ h^*) & \hookrightarrow & s[k=\dagger];\mathbf{handler}\{h^*\}\,E[v^n] & \\
 & & \text{if } ft = \text{cont}\ t_1^n \to t_2^* & \\
(\mathbf{handler}\{h^*\}\,v^*) & \hookrightarrow & v^* & \\
s;(\mathbf{handler}\{h^*\}\,E^x[v^n\,(\mathbf{suspend}\ x)]) & \hookrightarrow & s[+k=E^x];v^n\,(\mathbf{ref.cont}\ k)\,(\mathbf{br}\ l) & \\
 & & \text{if } h^* = h_1^*\,(\mathbf{on}\ x\ l)\,h_2^* \land x \notin h_1^* & \\
 & & \land\ s_{\text{tag}}[x] = t_1^n \to t_2^* & \\
s[k=E];v^n\,(\mathbf{ref.cont}\ k)\,(\mathbf{cont.bind}_{ft,ft'}) & \hookrightarrow & s[k=\dagger,+k'=E[v^n\,\_]];(\mathbf{ref.cont}\ k') & \\
 & & \text{if } ft\ = \text{cont}\ t^n\ t_1^m \to t_2^* & \\
 & & \land\ ft' = \text{cont}\ t_1^m \to t_2^* & \\
s[k=E];v^n\,(\mathbf{ref.cont}\ k)\,(\mathbf{resume\_throw}_{ft}\ x\ h^*) & \hookrightarrow & s[k=\dagger];\mathbf{handler}\{h^*\}\,E[v^n\,(\mathbf{throw}\ x)] & \\
 & & \text{if } s[x] = t^n \to \epsilon & \\
s[k=\dagger];(\mathbf{ref.cont}\ k)\,(\mathbf{resume}_{ft}\ h^*) & \hookrightarrow & s[k=\dagger];\mathbf{trap} & \\
s[k=\dagger];(\mathbf{ref.cont}\ k)\,(\mathbf{cont.bind}_{ft,ft'}) & \hookrightarrow & s[k=\dagger];\mathbf{trap} & \\
s[k=\dagger];(\mathbf{ref.cont}\ k)\,(\mathbf{resume\_throw}_{ft}\ x\ h^*) & \hookrightarrow & s[k=\dagger];\mathbf{trap} &
\end{array}
$$

Fig. 4. WasmFX Reduction

that they cannot contain any handlers for the given tag index x. The store holds function and tag declarations as well as allocated continuations. A frame consists of a sequence of local values. A continuation is given either by an evaluation context, or by the sentinel token dead when it is already used up.

The reduction rules are given in Figure 4 and are of the form $s_1;\mathit{ff}_1;e_1^* \hookrightarrow s_2;\mathit{ff}_2;e_2^*$. For brevity, we omit the store when it is not used; the same convention applies to frames. We write $\mathit{ff}[x=v]$ to denote the same frame as ff but with the value v assigned to local x. Similarly, $s[k=E]$ binds the continuation k to E in store s, and $s[+k=E]$ is the *extension* of s with a fresh continuation k.

The first set of rules involving locals, blocks, labels and branches, is once more inherited from plain Wasm, and we refer to Haas et al. [2017] for details. The only change here is that we allow branches to cross handlers, which uninstalls the handler. A call to a function introduces a suitable frame encapsulating the function body. Evaluation is performed inside a frame until it either results in values or throws.

The **cont.new** instruction extends the store with a fresh continuation that will immediately invoke the referenced function when it is resumed.

A continuation is resumed by installing a handler and inside it, reestablishing the evaluation context that represents the continuation. At the same time, the continuation is marked dead, so it cannot be resumed twice. A handler is removed once evaluation terminates with values. If it suspends with a tag for the handler, and there is no intervening handler for it in-between (expressed by the E^x context), then the appropriate number of tag arguments are extracted and the rest of the evaluation context up to the handler is reified into a freshly allocated continuation, arguments and continuation are pushed to the stack, and execution branches to the label associated with the tag.

Partially applying a continuation with **cont.bind** discards the old continuation in the store and adds a new one with the evaluation context extended with the supplied arguments.

Executing **resume_throw** also installs a handler, reestablishes the continuation's evaluation context, and marks the continuation dead. However, it inserts a **throw** instruction into the hole.

Finally, any attempt to consume a continuation that is already dead causes a trap.

3.4 Soundness

Type Preservation for Wasm [Haas et al. 2017; Watt et al. 2021] extends to the new instruction set. To prove this, we need to formulate adequate typing rules for the new administrative instructions:

$$\frac{s \vdash s_{\mathsf{cont}}(k) : t_1^* \to t_2^*}{s; C \vdash \textbf{ref.cont}\ k : \epsilon \to (\mathrm{ref\ cont}\ t_1^* \to t_2^*)} \qquad \frac{(C^\circ \vdash h : t^*\ \mathrm{clause})^* \qquad C^\circ \vdash e^* : \epsilon \to t^*}{C \vdash \textbf{handler}\{h^*\}\, e^* : \epsilon \to t^*}$$

Here, C° is the same as C but with all bindings for locals and labels removed. This restriction is necessary, because Wasm does not have closures. Without this restriction, we would allow dangling occurrences of local or label names to enter the store in the reduction rule for **suspend**. The restriction is preserved because all continuations start, via **cont.new**, from a separate function.

With that, and a couple of additional auxiliary judgements not shown here, we get:

THEOREM 3.1 (PRESERVATION). *If* $\vdash s; ff; e^* : t^*$ *and* $s; ff; e^* \hookrightarrow s'; ff'; e'^*$, *then* $\vdash s'; ff'; e'^* : t^*$.

For Progress, we need to define what the legal *results* of a computation are, namely, either multiple values, a trap, or an unhandled exception or suspension:

$$(\mathrm{result})\ \ r ::= v^* \mid \textbf{trap} \mid E[(\textbf{throw}\ x)] \mid E^x[(\textbf{suspend}\ x)]$$

Then we can prove, in the usual manner:

THEOREM 3.2 (PROGRESS). *If* $\vdash s; ff; e^* : t^*$, *then either* $e^* = r$, *or* $s; ff; e^* \hookrightarrow s'; ff'; e'^*$.

4 APPLICATIONS

4.1 Generators

When using generators, we deal with two concurrent computations. One sequentially consumes values while the other one produces (or yields) them one at a time. We can capture the production of values with a single tag $gen which carries the produced value as a parameter:

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

Then, we can write the following function which indefinitely produces natural numbers from 0:

```
(func $naturals
  (local $n i32)  ;; zero−initialised
  (loop $l
    (suspend $gen  (local.get $n))
    (local.set $n  (i32.add  (local.get $n)  (i32.const 1)))
    (br $l) ) )
```

Produced values can be consumed with a handler for $gen, for example one that sums up values until one exceeds $upto:

```
(func $sum_until (param $upto i32) (result i32)
 (local $v i32)
 (local $sum i32)
 (local $k (ref $cont))
 (local.set $k  (cont.new  (ref.func $naturals)))
 (loop $l
   (block $on_yield (result i32 (ref $cont))
     (resume $cont (on $yield $on_yield)  (local.get $k))
     (unreachable)  ;; $naturals never returns, so we can never get here
   ) ;;   $on_yield (result i32 (ref $cont))
   (local.set $k)  ;; store the new continuation for later resumption
   (local.set $v)  ;; store the produced value
   (local.set $sum  (i32.add  (local.get $sum)  (local.get $v)))  ;; add $v to $sum
   (br_if $l  (i32.lt_u  (local.get $v)  (local.get $upto)))
 (local.get $sum) )
```

Now, although the function $naturals is an infinite loop, we get 5050 by calling:

```
(call $sum_until  (i32.const 101))
```

Using a global table of active generators, we can implement a more ergonomic interface to generators, in which a programmer does not use handlers directly, but interacts solely through functions $init, which takes a continuation and returns a unique identifier, and $next, which takes the identifier and returns the next value.

4.2 Dynamic lightweight threads

We can make our lightweight threads functionality from Section 2 considerably more expressive by allowing new threads to be forked dynamically. For that, we declare a new $fork tag that takes a continuation as a parameter and (like $yield) returns no result.

```
(tag $fork (param (ref $cont)))
```

Thus, instead of manually enqueuing the threads, we can use the $fork tag:

```
(suspend $fork  (cont.new $cont  (ref.func $task1)))
(suspend $fork  (cont.new $cont  (ref.func $task2)))
```

Of course, we can obtain much more involved behaviour, as threads themselves may fork new ones. The scheduler only needs to be extended with a new clause for handling the $fork tag:

```
(func $scheduler (param $nextk (ref $cont))
 (loop $l
   (if  (ref.is_null (local.get $nextk))  (then (return)))
   (block $on_yield (result (ref $cont))
     (block $on_fork (result (ref $cont) (ref $cont))
       (resume $cont (on $yield $on_yield) (on $fork $on_fork)  (local.get $nextk))
       (local.set $nextk  (call $dequeue))
       (br $l) )
     ;; $on_fork, forkee and forker continuations on stack
     (local.set $nextk) ;; forker is next
     (call $enqueue)    ;; queue up forkee
     (br $l) )
     ;; $on_yield, yielder continuation on stack
   (call $enqueue)  ;; queue it up
   (local.set $nextk  (call $dequeue))  ;; take next
   (br $l) ) )
```

When handling a fork, there are two continuations on the stack. On top, we have the suspended continuation, and below it, the newly created thread that was given as an argument to $fork. We have three threads to choose from for the next step: the suspended active thread, the newly created thread, and the first inactive thread waiting in the queue. In the example above, we continue running the active thread and enqueue the new one, but we could easily adopt a different strategy.

4.3 Promises

We can adapt a similar approach for async/await-style *promises* [Bierman et al. 2012], in which one asynchronously runs a function that will compute a value, obtains an opaque promise, and then *awaits* that promise once the value is required. It is important to note that the function does not simply return the value, but uses it to *fulfill* a promise. If we represent promises with i32 identifiers, we can define a type of continuations that take and fulfill them:

```
(type $i32_func (func (param i32)))
(type $i32_cont (cont $i32_func))
```

We interface promises through four tags: $yield asynchronously yields as before, $fulfill takes a promise and a value to fulfill it with, $await takes a promise, awaits its fulfilled value, and returns it, and $async returns a new promise that will be fulfilled by the given continuation. We represent values with i64 in order to easily distinguish them from promise identifiers.

```
(tag $yield)
(tag $fulfill (param i32) (param i64))
(tag $await (param i32) (result i64))
(tag $async (param (ref $i32_cont)) (result i32))
```

Next, assume an external implementation of promises through externally defined functions $prom_* accessing a table that keeps a (perhaps unfulfilled) value for each promise and a continuation that awaits its result. Then, we can write a handler similar to the one for light-weight threads, except also handling the three additional tags. We describe the body of the four handlers; the overall structure mirrors that of the other examples.

When handling $fulfill, the stack contains the suspended continuation, the value, and the promise to be fulfilled. We keep the first as the active continuation $nextk and pass the other two to the external implementation, which returns the continuation awaiting the promise if any. If there is none, we do nothing, otherwise, we enqueue it as it is now unblocked.

```
(local.set $nextk)
(local.set $waiterk  (call $prom_fulfill))  ;; the call pops two operands off the stack
```

```
(if (i32.eqz (ref.is_null (local.get $waiterk))) ;; i32.eqz encodes negation
  (then (call $enqueue (local.get $waiterk))) )
```

When handling $await, the stack contains the current continuation and the promise it is waiting for. We first call the external function and check if the promise has already been fulfilled. If it has, then we can partially apply the current continuation and resume it. Otherwise, we attach the continuation to the promise and continue with the next thread in the queue.

```
(local.set $waiterk) (local.set $prom)
(if (call $prom_fulfilled (local.get $prom))
  (then
    (local.set $nextk
      (cont.bind $i32_cont $cont (call $prom_value (local.get $prom)) (local.get $waiterk)) ))
  (else
    (call $prom_await (local.get $prom) (local.get $waiterk))
    (local.set $nextk (call $dequeue)) ))
```

Finally, when handling $async, the stack contains the current continuation waiting for the new promise, and the asynchronous task that is meant to fulfill it. We create a new promise and pass it to both continuations; we enqueue the suspended continuation and run the new one.

```
(local.set $waiterk) (local.set $asynck)
(local.set $prom (call $prom_new))
(call $enqueue (cont.bind $i32_cont $cont (local.get $prom) (local.get $waiterk)))
(local.set $nextk (cont.bind $i32_cont $cont (local.get $prom) (local.get $asynck)))
```

4.4 Actors

As an example of handler composition, we consider Erlang-style *actors* [Armstrong et al. 1996]. These are independent processes that can spawn new actors and communicate to each other through mailboxes. We represent their interface through four tags: $send sends a message to a given mailbox, $receive receives the next incoming message, $spawn creates a new actor from a given continuation and returns its address, and $self returns the address of the current process. Like previously, we use i32 for addresses and i64 for messages.

```
(tag $send (param i64 i32))
(tag $self (result i32))
(tag $spawn (param (ref $cont)) (result i32))
(tag $recv (result i64))
```

To provide suitable type annotations, we need to define a number of additional continuation types:

```
(type $i64_func (func (param i64))) (type $i64_cont (cont $i64_func))
(type $i64_cont_func (func (param i64 (ref $cont)))) (type $i64_cont_cont (cont $i64_cont_func))
(type $cont_func (func (param (ref $cont)))) (type $cont_cont (cont $cont_func))
```

As an example, here is a function that receives a message and forwards it to actor $p:

```
(func $forward (param $p i32)
  (local $s i64)
  (local.set $s (suspend $recv))
  (suspend $send (local.get $s) (local.get $p)) )
```

And another that creates a chain of $n actors that forwards a message $m to the originating actor:

```
(func $chain (param $n i32) (param $m i64)
  (local $p i32)  ;; mailbox of the currently last actor in the chain
  (local.set $p  (suspend $self))
  (loop $l
    (if  (i32.eqz  (local.get $n))
      (then  (suspend $send  (local.get $m)  (local.get $p)))  ;; once done, send $m to the last actor
      (else
        (cont.new $i32_cont  (ref.func $forward))  ;; set up a new forwarding continuation
        (cont.bind $i32_cont $cont  (local.get $p)  ;; pass it the address of the currently last actor
        (local.set $p  (suspend $spawn))  ;; spawn a new actor and set it as the currently last
        (local.set $n  (i32.sub  (local.get $n)  (i32.const 1)))  ;; decrement $n and repeat
        (br $l) ) ) ) ) )
```

If an actor tries to receive a message, but its mailbox is empty, it must wait. The easiest way to implement this is by using threads as described in Section 4.2. We assume an interface to mailboxes through externally defined functions $mb_* and define a main function $act, which takes a continuation $k, creates a new mailbox and passes its address and the continuation to an auxiliary function $act_aux.

```
(func $act (param $k (ref $cont))
  (call $act_aux  (call $mb_new)  (local.get $k)))
```

The function $act_aux stores the mailbox address in a local $mine and resumes the continuation under a handler with the handler clauses as follows:

On $self, we take the existing continuation expecting the address, partially bind it to $mine, and store it as the next resumption.

```
(local.set $ik)
(local.set $nextk  (cont.bind $i32_cont $cont  (local.get $mine)  (local.get $ik)))
```

On spawning a new actor $you, we create a new mailbox, store its address in $yours, fork a new thread to run $you using $act_aux again, and pass the new address back to the continuation $ik.

```
(local.set $ik)  (local.set $you)  (local.set $yours (call $mb_new))
(suspend $fork
  (cont.bind $i64_cont_cont $cont  (local.get $yours)  (local.get $you)
    (cont.new $i64_cont_cont  (ref.func $act_aux)) ) )
(local.set $nextk  (cont.bind $i32_cont $cont  (local.get $yours)  (local.get $ik)))
```

On sending a message, we simply pass the tag argument (message) to the external function and resume:

```
(local.set $k)
(call $mb_send)
(local.set $nextk  (local.get $k))
```

Finally, on receiving a message, we first block until the mailbox $mine is no longer empty. Repeatedly, we check if it is empty, and if it is, we yield the control to other actors and try again later. Once the mailbox is non-empty, we read the message and pass it to the continuation $ik:

```
(local.set $ik)
(loop $blocked
  (if  (call $mb_empty  (local.get $mine))
    (then  (suspend $yield)  (br $blocked)) ) )
(local.set $nextk  (cont.bind $i64_cont $cont  (call $mb_recv  (local.get $mine))  (local.get $ik)))
```

Note that in the clauses for $spawn and $recv, we use tags $yield and $fork that provide an interface to threads and need to be handled. To run the actor $p, we first need to pass it to the $act handler. This needs to be converted to a continuation so that it can be further passed to the $scheduler handler as defined in Section 4.2.

```
(func $run_actor (param $p (ref $cont))
  (cont.new $cont_cont (ref.func $act)) ;; convert the $act function into a continuation
  (cont.bind $cont_cont $cont (local.get $p)) ;; partially apply the continuation to actor $a
  (call $scheduler) ) ;; handle fork & yield in the applied continuation
```

5 IMPLEMENTATION

We have implemented the full instruction set of WasmFX as an extension to the Wasm reference interpreter. Moreover, we have implemented a prototype of WasmFX in Wasmtime, a production-grade Wasm engine. In this section we describe the latter implementation. Wasmtime features an optimising just-in-time compiler for Wasm modules. We classify our implementation in Wasmtime as a prototype for three reasons. Firstly, at the time of writing we support only WasmFX programs on x86-64, whereas Wasmtime supports a much wider range of architectures including ARM64, ARM, RISC-V64, s390x, and x86. Secondly, we cannot implement **resume_throw** as Wasmtime does not yet support exceptions, thus we currently support only the other four instructions. We plan to implement **resume_throw** once support for exceptions lands in Wasmtime. Thirdly, and most importantly, our implementation piggybacks on the existing fiber API in Wasmtime [Crichton 2021], which enables the Wasm host to run functions asynchronously. Targeting the fiber API allows us to quickly prototype the design in a production-grade setting, and its implementation integrates well with standard debugging and profiling tools as it already emits the necessary information to construct DWARF unwind tables. On the other hand, it has the problem that fibers exist *outside* the Wasm world, meaning that every interaction with a fiber necessarily needs to call out to the host, a relatively expensive operation. Moreover, it requires us to *box* continuation parameters and results, because the host function cannot be polymorphic over continuation types.

The Fiber API. Figure 5a shows the interface to the Wasmtime fiber API. The FiberStack::new method allocates a new stack of minimum size bytes. Figure 6 details the actual stack layout. The first 32 bytes are reserved for the header. The initial 8 bytes are used to store the payload for resumes, suspends, and returns. The next 8 bytes are used for the local stack pointer. The next 8 bytes store the pointer to the parent fiber. These bytes were added by us to implement the dynamic scoping of handlers. Finally, there is a guard page to detect stack overflow. We also extend the instance context of Wasmtime with an additional field to keep track of the currently executing fiber such that we can retrieve its pointer when resuming and suspending.

The key entity is the Fiber structure, which is parameterised by three type variables for resume payloads (R), suspend payloads (S), and return values (A). We instantiate R and A to the unit type, because we store payloads directly in the bespoke buffer on the fiber stack. The reason for using this approach is that we do not a priori which types the user provided program will use. Consequently, we are forced to box everything that crosses the Wasm-Host boundary and vice versa. Meanwhile, we instantiate S to u32 as we use it to communicate index of the control tag supplied to **suspend**.

Fiber::new takes a fresh FiberStack, and the suspended computation to run on the said stack. It installs the launchpad code necessary to run the computation when the initial Fiber::resume occurs. The Fiber::resume method suspends the currently executing fiber and continues execution of the provided Fiber. The Suspend::suspend method works similarly.

```
trait FiberStack {
  fn new(size: usize) -> io::Result<Self>
}
trait<R, S, A> Fiber<R, S, A> {
  fn new(stack: FiberStack,
           func: FnOnce(R, &S<R, S, A>) -> A
  fn resume(&self, val: R) -> Result<A, S>
}
trait Suspend<R, S, A> {
  fn suspend(&self, S) -> R
}
```

(a) The Essence of the Wasmtime Fiber API

```
.wasmtime_fiber_switch:
  // Save callee-saved registers
  push ...
  // Load resume pointer, save previous
  mov rax, -0x20[rdi]
  mov -0x20[rdi], rsp
  // Swap stacks
  mov rsp, rax
  pop ... // restore callee-saved registers
  ret
```

(b) Fiber Switching x86_64 Assembly Code

Fig. 5. Wasmtime Fiber API and Context Switching Code

Stack Switching. Stack switching in the fibers library is implemented in x86_64 assembly code. Figure 5b shows the stack switching routine. It pessimistically saves any necessary registers to the stack. The pointer to the stack is passed in register rdi, from which we load the saved rsp (local stack pointer); subsequently we store the current stack pointer in the same header spot. The actual stack swapping occurs by overriding the value of rsp with the previously loaded value. After restoring saved registers, the top of the stack holds the return location saved when the fiber was suspended; a return instruction is executed, continuing execution from the suspended point.

Compiling WasmFX to Wasmtime Fiber. The translation from WasmFX continuation instructions to Wasmtime fibers is mostly straightforward. We implement each instruction as a *libcall* which calls from compiled Wasm code to Wasmtime runtime code. We realise each instruction as follows.

We map **cont.new** to Fiber::new. We box the resulting fiber so we can null out the continuation object once it has been supplied to **resume**. The suspended computation merely marshals values to the provided Wasm function. For **cont.bind** we write the payload directly to the heap-allocated result cell and return a new continuation object (containing the same fiber reference).

The handler clauses on the **resume** instruction get compiled to the core Wasm instruction **br_table**, which is a jump table. The table jumps to handler blocks for tags that are handled, and re-suspends to the parent (if any) for tags that are not. The libcall for **resume** boxes the arguments, sets the parent pointer of the fiber to the currently executing fiber, and invokes Fiber::resume. We return the result of this function to the handler in Wasm, distinguishing return from suspend with a sentinel value.

The libcall for **suspend** replaces the current stack pointer in the context with its parent. It then writes the payload, including the name of the tag, to its result cell and transfers control.

5.1 Experiments

We perform some preliminary experiments to gather data to obtain insight into our prototype implementation of WasmFX fares against the state-of-the-art. We perform two experiments: 1) we measure the performance characteristics, i.e. binary size, run time, and memory performance on a micro benchmark provided by Leijen and Sivaramakrishnan [2021]; 2) we compare the binary size of programs produced by the TinyGo compiler with and without WasmFX as a backend. The experiments are conducted on an AMD Ryzen 9 5900X 12-core 3.75GHz CPU with 32 GB memory powered machine running Ubuntu 22.04 LTS.

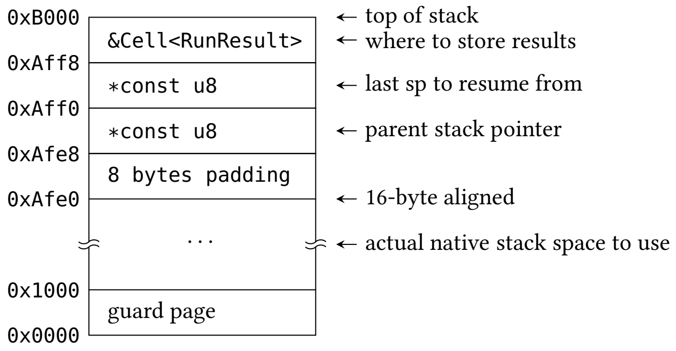

Fig. 6. Fiber Stack Layout

The first benchmark, provided by Leijen and Sivaramakrishnan [2021], simulates a web server workload. It is written in C. The benchmark runs 10000 coroutines concurrently (each intended to represent a unique http request), each coroutine simply suspends to simulate I/O and then performs some stack-heavy computation. The benchmark performs 10 million requests, meaning it spawns 10 million coroutines in total.

We run the benchmark in three ways: 1) using WasmFX; 2) Asyncify; and 3) a bespoke hand-written state machine transformation of the benchmark program. It is worth noting that the WasmFX and Asyncify approaches require no programmer intervention, whereas the latter approach requires a complete rewrite of the program. We run each benchmark program five times and measure the size of the compiled Wasm binary, median wall clock time, and maximum physical memory usage. We use Wasmtime to run each benchmark. To compile the C program to Wasm we use `clang` version 14 with optimisation flag `-O3` and the binaryen toolchain version 105. For the experiments, we modify the Wasmtime implementation of WasmFX to allocate fiber stacks using malloc instead of mmap as this is the memory allocation scheme used by the Asyncify and hand-written program. In particular, we use the optimised `mimalloc` allocator [Leijen et al. 2019] rather than the default system allocator. Finally, WasmFX and Asyncify allocate stacks with the fixed size of 4096 bytes.

Figure 7a reports the results for the first benchmark. The space performance of WasmFX is on par with the state-of-the-art (optimised Asyncify) in both binary file size and memory usage. Though, the binary file size is better when compiling with WasmFX. However, the run time performance of the WasmFX implementation is more than a factor of four slower. Performance analysis indicates that about 37% of samples are spent in `wasmtime_fiber_switch` (Figure 5b) with an additional 33% of samples in the Rust libcall code supporting the continuation instructions, which are both highly unoptimised. Saving registers, calling between the host and Wasm, and boxing values likely accounts for much of this cost. As expected the bespoke implementation is fastest and exhibits the best memory utilisation of all. The primary reason for the space efficiency is that it does not allocate 4096 bytes stack for each coroutine.

In the second experiment we measure the potential effect that WasmFX and Asyncify have on the Wasm binary size. Specifically, we compile two off-the-shelf Go micro-benchmark programs, both of which makes heavy use of coroutines [KJP 2019]. We compare the size of the Wasm binary generated by the TinyGo compiler with WasmFX as a backend and optimised Asyncify as a backend. For the WasmFX backend, we use a coroutine scheduler which closely resembles that of

	Binary Size	Wall Time	Memory Usage
WasmFX	24 KB	2310 ms	63.9 MB
Asyncify	36 KB	480 ms	64.8 MB
Bespoke	27 KB	140 ms	18.9 MB

(a) Performance Results for WasmFX, Asyncify, and Bespoke

	main-kjp.go	coroutines.go
WasmFX	156 KB	7.2 KB
Asyncify	597 KB	40 KB

(b) Binary Size Comparison for TinyGo Programs

Fig. 7. Preliminary Results

Section 2. The results are present in Figure 7b. The compiled size of the WasmFX powered binary is significantly smaller than the state-of-the-art.

6 DISCUSSION

6.1 Design Considerations

In this section we elaborate on the motivations for the design choices of WasmFX.

Direct-style. The goal of WasmFX is to provide a universal mechanism for implementing non-local control features. So-called "stackless" approaches such as Asyncify [Zakai 2019] simulate such features on top of Wasm by applying some form of global transformation into continuation-passing style or a state machine [Bierman et al. 2012]. To support non-local control flow, functions must be instrumented, resulting in both runtime cost and increase in code-size. The runtime cost can be alleviated by compiling functions twice, once with support for non-local control flow and once without, but that increases code size further. WasmFX provides non-local control natively in direct style, providing a convenient target for features such as actors, async/await, green threads, etc.

Delimited control. It is well-established that delimited control operators provide a universal mechanism for implementing non-local control features. Effect handlers provide a structured form of delimited control supporting fine-grained definition and modular composition of non-local control features [Kammar et al. 2013]. Recent work has shown that effect handlers can be implemented efficiently in practice [Ghica et al. 2022; Sivaramakrishnan et al. 2021]. A key difference between effect handlers and traditional delimited control operators is that the handling of a continuation occurs at the delimiter rather than the capture-site of the continuation. In this sense, effect handlers are similar to Sitaram's run/fcontrol operator [Sitaram 1993]. Though, effect handlers provide multiple typed control tags with implicit effect forwarding, whereas run/fcontrol provides a single universal control tag, which in turn means there is no notion of implicit effect forwarding. Instead each instance of run is indexed such that an invocation of fcontrol can be explicitly directed to a particular run instance (for more details, the interested reader may consult [Hillerström 2021, Appendix A] for a comprehensive survey of first-class control operators).

Why not undelimited control? It is possible to take undelimited control abstractions such as call/cc [Sperber et al. 2009] or lightweight threads as primitive. It is even possible to build general delimited control abstractions, including effect handlers [Ghica et al. 2022], on top of these. However, undelimited continuations do not compose, which makes working with undelimited control operators significantly more difficult in practice [Kiselyov 2012]. The composability of delimited control is what underpins the support for structured concurrency features in Java 19, for instance [Bateman and Pressler 2021; Pressler 2018].

Synergy between the stack typing discipline of Wasm and effect handlers. A further advantage of basing our design on effect handlers is that the use of typed control tags fits seamlessly with the stack typing discipline of Wasm. When suspending or resuming it is natural to combine the associated context switch with the transfer of data. Control tags allow for a choice of different kinds of suspension each with their own type of payload which is passed unboxed on the stack. Moreover, the result types of the control tag allow the resuming of the suspension to pass back its own data unboxed on the stack. The coupling of the parameter and result types of a control tag provides a statically typed coupling between a suspension and its corresponding resumption.

One-shot continuations support key use-cases and efficient implementation. The WasmFX design naturally extends to support multi-shot continuations, but the key use-cases we are targeting (primarily different forms of concurrency) require only one-shot continuations. One-shot continuations admit direct and efficient implementations of continuations as stacks in which it is never necessary to copy the stack. In principle, it would be possible to extend Wasm with an affine type system to statically ensure that continuations are invoked at most once, but this would place an undue burden on producers and validators of Wasm code, so we doubt it would be realistic. Instead, we follow the lead of OCaml 5 and dynamically check that continuations are invoked at most once — trapping in the case that an attempt is made to invoke a continuation more than once.

Avoiding cycles. Wasm currently includes no native support for automatic memory management. There is a garbage collection proposal, but some producers may prefer to not use it and some engines may not be able to support it. WasmFX is consciously designed to avoid any dependency on garbage collection. In particular, it supports an implementation that avoids cycles in the heap and can thus be implemented using plain reference counting. This property depends crucially on the affine semantics of continuations: creating a cycle is possible only if a continuation can get a handle to itself and stores it on its own stack. However, a continuation can only receive data when being resumed. And once resumed, a continuation is immediately marked dead. An implementation could hence null it out immediately, cutting the reference from continuation to stack. Consequently, even if a continuation is passed to itself as a reference upon resumption, it will only receive a dead reference that no longer points anywhere, preventing the possibility of forming a cycle.

Avoiding allocation. A downside of the implementation that avoids cycles is that it relies on allocating a fresh continuation object every time we **suspend** or partially apply a continuation with **cont.bind**. Perhaps, counter-intuitively, an implementation — e.g., one with GC — can trade the no-cycles property for an allocation-free implementation, where **suspend** and **cont.bind** can reuse the same physical continuation object each time, or in fact, merge it with the stack itself. The problems with doing this naively are twofold: 1) each **suspend** or **cont.bind** can change the type of the underlying continuation, and we must maintain type soundness; 2) we need to distinguish between these legitimate reuse and erroneous attempts of using a continuation twice — i.e., we need to be able to distinguish live and dead references to the same physical continuation object.

A solution to both these problems is to augment the representation of continuation references and objects. We can represent a continuation reference by a *fat pointer* consisting of a pair of a pointer to the continuation object and a unique sequence counter. The continuation object itself also includes a sequence counter. Each time it's consumed, the sequence counter of the object is incremented. If there is a mismatch between the sequence counter of a continuation reference and the object it points to, then this constitutes a linearity violation and we trap. The sequence counter should at least be a 64-bit integer as it is not unrealistic to expect a real program to suspend more than 2^{32} times. Under this implementation approach, heap-stack cycles *can* arise. Consider:

```
(type $f (func (param (ref $c)))
(type $c (cont $f))
(tag $pause)
(func $task (param $x (ref $c)))
  (suspend $pause) )
(func $run
  (local $c (ref $c))
  (local.set $c  (cont.new $c  (ref.func $task)))
  (block $on_pause (result ...)  (resume (on $pause $on_pause)  (local.get $c)  (local.get $c)))
  ... );; here we get a contin. object that physically is the same as $c AND whose stack has a pointer to $c
```

If, on the other hand, **suspend** allocates a fresh continuation object, and the previous one is nulled out, then no cycle will exist. WasmFX's design, thanks to linear continuations, gives implementations a choice between these different trade-offs. This choice is semantically transparent to Wasm code.

Another, more disruptive design would involve separating out the sequence counter as its own special capability object. This would have the advantage of avoiding the need for fat pointers at the cost of having to change the interface to our instructions.

Unityped delimited continuations. The bespoke Wasmtime Fiber API can be cast as an instance of delimited control. Coincidentally, it provides operators with the same names as the core instructions of WasmFX: **new**, **resume**, and **suspend**, which also behave quite similarly. In essence it provides a unityped form of delimited continuations, where the payload types for suspending and resuming are fixed, so in order to support different types of payload the payload cannot be unboxed on the stack. Instead of using a tag to determine where to handle a suspend, it must always be handled by the immediate parent context. This means there is no need to build any kind of special handler construct into the syntax. Because payload types are fixed, the type of the current continuation object does not change each time it is resumed. This means that it can be reused in a type safe way without the need for fat pointers or a linearity check, at the cost of having to box heterogeneous payloads and not being able to support an implementation that rules out cycles.

Combining handling with resumption. Traditional accounts of effect handlers decouple handling of computations from resuming of continuations. The WasmFX design instead couples handling with resumption offering several advantages: 1) it allows a different handler to be installed each time a continuation is resumed; 2) it allows for the handling component of the **resume** instruction to be rather concise as it is simply a mapping from tags to labels — the actual handler code is attached to the block structure defined by the labels; 3) it makes implementation of stack segments simpler and more uniform as handlers are now in one-to-one correspondence with active continuations. Deep handlers [Plotkin and Pretnar 2009] automatically rewrap the current handler around the resumption. Shallow handlers [Hillerström and Lindley 2018; Kammar et al. 2013] instead require the handler to be reinstalled each time a continuation is resumed. Deep handlers are easier to reason about and optimise, but shallow handlers are more convenient in some cases. The handlers in WasmFX can be seen as a hybrid of shallow and deep: "sheep handlers". As with deep handlers the body of a continuation is guaranteed to be wrapped in some handler. As with shallow handlers this handler can be changed each time we resume.

Return clauses. Traditional effect handlers include an explicit return clause which allows the handler to perform some transformation on the final value returned by a computation. Return clauses are directly inspired by and offer similar advantages to the success continuation of Benton and Kennedy's [Benton and Kennedy 2001] exceptional syntax variation of exception handlers. WasmFX primitives on the other hand are lower level. Because there is not automatically a join point between the code following a **resume** and the code following a handler clause it is easy to wire in code to simulate a return clause. When we do want to join up the control flow then we do so explicitly with a function call or a branch instruction.

Consider a basic deep effect handler for a computation that returns an integer which handles an operation ask returning an integer and has a return clause that converts the final result into a floating point number.

$$\begin{aligned} &\textbf{return}\ x \mapsto \mathit{intToFloat}\ x \\ &\mathsf{ask}\ k \qquad \mapsto k\ 42 \end{aligned}$$

We might implement this in WasmFX as follows

```
(type $icont (cont (func (result i32)))
(tag $ask (result i32))
(block $h  (result f32)
  (loop $l
    (block $on_ask (result i32 (ref $icont))
      (resume $cont (on $ask $on_ask)  (local.get $k))
      (f32.convert_i32_s) ;; return clause
      (br $h)
    ) ;; on_ask
    (cont.bind $icont $cont  (i32.const 42))
    (local.set $k)
    (br $l) ) )
```

where $k is initialised with the computation to be handled. Resuming the continuation leaves an **i32** value on the stack which the image of the return clause (**f32.convert_i32_s**) converts into a **f32**.

Its worth noting that in the typing rule for **resume** (Figure 2) the types t_2^* always represent the original result types of the computation being handled and not the final result type after handling.

In the case of a scheduler, for instance, the code that dequeues the next continuation and runs it constitutes the return clause. Though in that case the return clause is fixed, because WasmFX handlers are sheep handlers the return clause can change each time a continuation is resumed. Its worth noting that in the typing rule for **resume** (Figure 2) the types t_2^* always represent the original result types of the computation being handled and not the final result type after handling.

6.2 Extensions

We strived to start with a minimal design for WasmFX, but of course, many extensions are possible.

Named handlers. The core WasmFX design captures a continuation by suspending as far as the nearest enclosing handler that matches the specified tag. An alternative is to suspend to a specific named handler. We can support such a mechanism by adding a new reference type handler t^* along with special variants of **resume** and **suspend**. The **resume_with** instruction is just like **resume** except that it passes a fresh handler name to the continuation.

$$\frac{ft = \ t_1^*\ (\mathsf{ref\ handler}\ t_2^*) \to t_2^* \qquad (C \vdash h : t_2^*\ \mathsf{clause})^*}{C \vdash \textbf{resume\_with}\ h^* : t_1^*\ (\mathsf{ref\ cont} ft) \to t_2^*}$$

The **suspend_to** instruction is just like **suspend** except it takes an additional handler argument.

$$\frac{C_{\mathrm{tag}}(x) = t_1^* \to t_2^*}{C \vdash \textbf{suspend\_to}\; x : t_1^*\ (\mathrm{ref}\ ht) \to t_2^*}$$

A handler reference is similar to a prompt in a system of multi-prompt delimited continuations [Gunter et al. 1995]. However, since it is created fresh for each handler, multiple activations of the same prompt cannot exist by construction. The ergonomic tradeoffs between named and unnamed handlers are reasonably apparent in high-level source languages, but less so for a low-level target language like Wasm. Unnamed handlers incorporate a form of dynamic binding, enabling lightweight composition of effects. A natural way to simulate this dynamic binding involves threading concrete handler implementations through a program, something which can be painful to do manually, impeding modularity. Named handlers offer a form of generativity supporting a form of effect encapsulation [Ghica et al. 2022]. Unnamed and named handlers, like single-prompt and multi-prompt delimited control, can simulate one another (at a cost). In the future we plan to measure that cost in Wasm. We suspect it may ultimately be worthwhile to support both as some higher-level systems do [Ghica et al. 2022; Xie et al. 2022].

Barriers. In some situations — for example, when interacting with legacy code that cannot or does not expect to be suspended and have interleaved execution — it is desirable to prevent suspensions beyond a certain extent. For this purpose, a block-like instruction **barrier** e^* can easily be introduced that bars any suspension from inside e^* across its boundary. A barrier may simply be viewed as a "catch-all" handler that handles any control tag by immediately trapping.

Multi-shot. Continuations in WasmFX are one-shot. Some applications of effect handlers such as backtracking, probabilistic programming, and process duplication exploit multi-shot continuations, but the key use-cases we have in mind do not, and restricting attention to one-shot continuations allows us to avoid having to copy stacks. Nevertheless, it is natural to envisage a future extension that includes support for multi-shot continuations by way of a continuation clone instruction. However, some Wasm engines would have a hard time with such an extension, since they use heterogeneous stacks mixing Wasm with C++ frames, which cannot easily be moved or copied.

Tail-resumptive handlers. A handler is said to be tail-resumptive if it invokes the continuation in tail-position in every handler clause [Xie and Leijen 2021]. The canonical example of a tail-resumptive handler is dynamic binding. The handler clauses of a tail-resumptive handler can be inlined at the suspend sites, because they do not perform any non-trivial control flow manipulation, they simply retrieve a value. Inlining clauses means that no time is spent constructing continuation objects. WasmFX as it stands includes no facilities for identifying and inlining tail-resumptive handlers. Moreover, the primary motivation for the design for WasmFX is to support use-cases in which continuations are *not* invoked immediately. Nevertheless, it is natural to envisage a future iteration of this proposal that includes an extension for distinguishing tail-resumptive handlers.

6.3 Related work

Delimited control in Wasm. Wasm/k [Pinckney et al. 2020] was an early attempt to add first-class continuations to WebAssembly 1.0. Unlike WasmFX, it does not support multiple named control tags, which are necessary to support several distinct uses of non-local control in a typed and modular way. Moreover, since Wasm/k did not consider emerging features of WebAssembly 2.0, it does not compose with features that are now part of the standard, such as typed function references, and others that are in the process of being standardised, such as exceptions.

One-shot continuations. Using the call stack for implementing one-shot continuations has a long history. Bruggeman et al. [Bruggeman et al. 1996] show how to implement one-shot continuations using segmented stacks in Scheme. Farvardin et al. [Farvardin and Reppy 2020] perform a comprehensive evaluation of various implementation strategies on modern hardware including several stack-based implementations of one-shot continuations such as contiguous, segmented, and resizable stacks, as well as representing continuations using a continuation-passing style transformation in the compiler. The paper shows that if the primary concern is sequential performance with advanced control-flow features, then contiguous or resizable stacks are the best strategy.

Growing stacks. Resizeable segmented stacks are also used in the OCaml implementation for delimited continuations [Sivaramakrishnan et al. 2021]. The OCaml managed stack starts out small but when the stack would overflow it is reallocated – potentially to a different location. This is safe in OCaml as the compiler and runtime ensure that there are never any pointers into the stack. This is generally not the case though in most languages, like C and C++, and growing stacks by reallocation cannot be used in Wasm to implement effect handlers.

The recent *libmprompt* library [Leijen and Sivaramakrishnan 2021] enables segmented stacks at a system level where it can grow stack segments *in-place* by reserving virtual address space upfront, but committing the memory on-demand when the stack would overflow. This can be very efficient and could potentially be used for the WasmFX implementation in Wasmtime.

Async/Await and Generators. Various languages, like C++, Javascript, C#, Rust, etc., implement specific instances of delimited control in the form of async/await and generators. To compile such delimited control without runtime support (like WasmFX would provide!), these languages require the async or generator functions to be annotated, and then compile those functions in a special way that allows them to be suspended on an await or yield. Generally each such function allocates its stack frame in the heap, together with a state machine that can resume at each await/yield point [Bierman et al. 2012]. It is possible to compile this efficiently without allocation for small inlinable functions but with increased nesting the overhead can be quite large compared to having runtime support for direct stack switching. Moreover, due to the special calling convention it is inherently not compositional where one needs to decide upfront whether a function can be async or not [Nystrom 2015]. Xie and Leijen [2021] show how one can compile general effect handlers using a monadic approach which avoids the need for an explicit state machine and can use the standard C call stack, where only on a suspend the stack is reified to an explicit continuation. This approach can be quite efficient but comes at the cost of expanding the generated code at least by a factor 2 and, as with async/await and generators, adds a complex compilation step.

Continuation marks. Chez Scheme supports continuation marks [Flatt and Dybvig 2020], a language feature that supports stack inspection on top of which features such as exceptions, debuggers and profilers are implemented in the presence of first-class continuations. Flatt and Dybvig [2020] note that implementation strategy for continuation marks and multi-prompt delimited continuations are similar.

6.4 Future work

There are many opportunities for optimising the prototype implementation. Here we outline some low-hanging fruit that we plan to explore next. First, calling out to the Rust runtime code incurs an overhead. Instead, the **suspend** and **resume** instructions could be compiled directly to architecture-specific instructions (Figure 5b) which manipulate the registers, headers, and stack pointer. Second, all values passed in **resume** and **suspend** are currently stored and loaded from memory. For performance, these values could be passed directly, observing the calling conventions of the host system. Finally, the fixed-sized system stacks induce allocation burden that may be unnecessary: experiments in other languages have found that most continuations require little stack memory [Sivaramakrishnan et al. 2021]; other allocation techniques such as those used by libmprompt [Leijen and Sivaramakrishnan 2021] may perform better.

We described a design and implementation for non-local control flow in Wasm based on effect handlers. As such, WasmFX provides a unified and composable extension that can directly express a wide variety of rich control flow constructs found in various languages. We are looking forward to strengthening our prototype implementation in Wasmtime. Furthermore, we plan to add back-ends to various languages with rich control flow to directly target the new WasmFX instructions.

REFERENCES

Heejin Ahn. 2022. Exception Handling Proposal for WebAssembly. https://webassembly.github.io/exception-handling/ Accessed 2022-10-27.

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in Erlang, Second Edition.* Prentice Hall International, Hertfordshire, UK.

Alan Bateman and Ron Pressler. 2021. JEP 428: Structured Concurrency (Incubator). https://openjdk.org/jeps/428. Accessed 2023-04-14.

Nick Benton and Andrew Kennedy. 2001. Exceptional Syntax Journal of Functional Programming. *J. Funct. Program.* 11, 4 (2001), 395–410.

Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' Play: Formalizing Asynchronous C#. In *ECOOP (Lecture Notes in Computer Science, Vol. 7313).* Springer, 233–257.

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *PLDI.* ACM, 99–107.

Alex Crichton. 2021. Wasmtime Fiber API. https://docs.wasmtime.dev/api/wasmtime_fiber/index.html. Accessed 2023-04-14.

Kavon Farvardin and John H. Reppy. 2020. From folklore to fact: comparing implementations of stacks and continuations. In *PLDI.* ACM, 75–90.

Matthew Flatt and R. Kent Dybvig. 2020. Compiler and Runtime Support for Continuation Marks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/3385412.3385981

Dan R. Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1639–1667.

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95).* ACM, 12–23. https://doi.org/10.1145/224164.224173

Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J.F. Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Notices* 52, 6 (June 2017), 185–200.

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS (Lecture Notes in Computer Science, Vol. 11275).* Springer, 415–435.

Daniel Hillerström. 2021. *Foundations for Programming and Implementing Effect Handlers.* Ph. D. Dissertation. School of Informatics, The University of Edinburgh, Scotland, UK.

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

Oleg Kiselyov. 2012. An argument against call/cc. https://okmij.org/ftp/continuations/against-callcc.html Accessed 2023-04-12.

KJP. 2019. Benchmarking 1 million C# tasks vs Go goroutines: Is there any difference? https://karl-pickett.medium.com/benchmarking-a-toy-c-task-vs-a-go-goroutine-is-there-any-difference-248f73f7f7b7

Daan Leijen and KC Sivaramakrishnan. 2021. libmprompt. https://github.com/koka-lang/libmprompt Accessed 2023-04-14.

Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. In *APLAS (Lecture Notes in Computer Science, Vol. 11893).* Springer, 244–265.

Bob Nystrom. 2015. What Color is Your Function? https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

Donald Pinckney, Arjun Guha, and Yuriy Brun. 2020. Wasm/k: delimited continuations for WebAssembly. In *DLS.* ACM, 16–28.

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP (Lecture Notes in Computer Science, Vol. 5502).* Springer, 80–94.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013).

Ron Pressler. 2018. Project Loom: Fibers and Continuations for the Java Virtual Machine. https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html. Accessed 2023-04-14.

Andreas Rossberg. 2019. WebAssembly (Release 1.0). https://webassembly.github.io/spec/ Accessed 2020-01-01.

Andreas Rossberg. 2022. Function Reference Types Proposal for WebAssembly. https://webassembly.github.io/function-references/ Accessed 2022-10-27.

Andreas Rossberg. 2023. WebAssembly (Release 2.0). https://webassembly.github.io/spec/ Accessed 2023-20-02.

Dorai Sitaram. 1993. Handling Control. In *PLDI*. ACM, 147–155.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. ACM, 206–221.

Michael Sperber, Kent R. Dybvig, Matthew Flatt, Anton van Stratten, Robby Bruce Findler, and Jacob Matthews. 2009. Revised^6 Report on the Algorithmic Language Scheme. *J. Funct. Program.* 19, S1 (2009), 1–301.

Guy L. Steele Jr. 1990. *Common LISP: The Language (2nd Ed.)*. Digital Press.

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Rao, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods: 24th International Symposium*. Springer-Verlag, Berlin, Germany, 61–79.

Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-Class Names for Effect Handlers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 126 (oct 2022), 30 pages. https://doi.org/10.1145/3563289

Ningning Xie and Daan Leijen. 2021. Generalized Evidence Passing for Effect Handlers (or, Efficient Compilation of Effect Handlers to C). *Proc. ACM Prog. Lang. (ICFP'21)* 5, ICFP (Aug. 2021), 71.

Jeremy Yallop. 2023. A collaborative biliography of work related to the theory and practice of computational effects. https://github.com/yallop/effects-bibliography Accessed 2023-04-14.

Alon Zakai. 2019. Pause and Resume WebAssembly with Binaryen's Asyncify. https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html Accessed 2022-10-27.