Wasm/k: Delimited Continuations for WebAssembly

Donald Pinckney
pinckney.d@northeastern.edu
Northeastern University
USA

Arjun Guha
a.guha@northeastern.edu
Northeastern University
USA

Yuriy Brun
brun@cs.umass.edu
University of Massachusetts Amherst
USA

Abstract

WebAssembly is designed to be an alternative to JavaScript that is a safe, portable, and efficient compilation target for a variety of languages. The performance of high-level languages depends not only on the underlying performance of WebAssembly, but also on the quality of the generated WebAssembly code. In this paper, we identify several features of high-level languages that current approaches can only compile to WebAssembly by generating complex and inefficient code. We argue that these problems could be addressed if WebAssembly natively supported first-class continuations. We then present Wasm/k, which extends WebAssembly with delimited continuations. Wasm/k introduces no new value types, and thus does not require significant changes to the WebAssembly type system (validation). Wasm/k is safe, even in the presence of foreign function calls (e.g., to and from JavaScript). Finally, Wasm/k is amenable to efficient implementation: we implement Wasm/k as a local change to Wasmtime, an existing WebAssembly JIT. We evaluate Wasm/k by implementing C/k, which adds delimited continuations to C/C++. C/k uses Emscripten and its implementation serves as a case study on how to use Wasm/k in a compiler that targets WebAssembly. We present several case studies using C/k, and show that on implementing green threads, it can outperform the state-of-the-art approach Asyncify with an 18% improvement in performance and a 30% improvement in code size.

CCS Concepts: • **Software and its engineering → Coroutines**.

Keywords: virtual machines, first-class continuations, formal language semantics

DLS '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8175-8/20/11…$15.00
https://doi.org/10.1145/3426422.3426978

ACM Reference Format:
Donald Pinckney, Arjun Guha, and Yuriy Brun. 2020. Wasm/k: Delimited Continuations for WebAssembly. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3426422.3426978

1 Introduction

For decades, ECMAScript (JavaScript) was the only programming language that was universally supported by all major web browsers. There are now several, high-performance JavaScript implementations that make it possible to run large programs, such as spreadsheets, IDEs, and video editors on the Web. In fact, many contemporary desktop applications, such as Slack and Visual Studio Code, are now built with JavaScript and other web technologies [2].

Since web browsers, and thus JavaScript, are ubiquitous, there are now scores of programming languages with compilers that emit JavaScript to run on the Web. However, compiling to JavaScript has two serious drawbacks: 1) programs may perform poorly when compiled to JavaScript, and 2) a variety of language features, such as threads, are hard to compile to JavaScript.

However, there is now an alternative to JavaScript. *WebAssembly* [18] is a recently introduced low-level language that aims to be a better compiler target language than JavaScript. All modern web browsers support WebAssembly, and despite its name, there are several WebAssembly runtime systems that are not embedded in browsers. When programs written in C/C++ are compiled to WebAssembly, they run 1.3× faster on average than when they are compiled to JavaScript [18, 20]. However, given WebAssembly as it exists today, it remains difficult to compile a variety of language features, including green threads, coroutines, and continuations. In fact, many languages that support these features natively, either do not support them in WebAssembly, or produce slow code. For example, the Go compiler has a WebAssembly backend. However, it struggles to support green threads (*Goroutines*), which makes the compiler difficult to maintain, and produces code that performs poorly [5–7, 9].

Safety is a key design goal of WebAssembly, which is necessary for web browsers to run untrusted code in a trustworthy manner. Toward this end, WebAssembly programs are isolated from the browser, and cannot directly alter the low-level state of the WebAssembly runtime. In particular, the WebAssembly stack is not stored on the WebAssembly heap.

Moreover, WebAssembly only supports structured control-flow and does not support exceptions[1], *goto*, and *longjmp*. These restrictions make WebAssembly validation straightforward and fast. However, they make it difficult to implement non-local control flow. For example, Goroutines and green threads require low-level support for switching between stacks, which WebAssembly does not directly support.

State of the art. To workaround the restrictions of WebAssembly, the Go compiler performs a global program transformation, which 1) builds a copy of the WebAssembly stack in the heap to store local variables, and 2) simulates non-local jumps via an elaborate state machine in each function. Asyncify [28] uses a similar approach to add virtual instructions that save and restore stacks to WebAssembly. Prior benchmarks indicate that Asyncify has a performance overhead of 20%–100%, and a similar increase in code size [28]. Moreover, since these tools use a *global transformation* to achieve non-local control flow, programmers are forced to pay a steep cost for such features, even when their code uses them minimally.

Our contributions. In this paper, we present `Wasm/k`, an extension to WebAssembly that adds support for *first-class, delimited continuations*, which are sufficient to implement a wide variety of language features, including green threads, coroutines, and exceptions. Our extension has only a handful of new instructions, is designed to support efficient implementation, and is designed to work well when WebAssembly programs interact with other languages (e.g., JavaScript).

Although first-class continuations are a well-known abstraction, they are typically found in higher-level programming languages (e.g., Scheme and Racket). These languages are compiled to low-level native code that does not support first-class continuations. Our work inverts this tradition and instead adds first-class continuations directly to a low-level language. In doing so, our design tackles unique challenges imposed by the low-level setting, such as the lack of first-class functions, lack of garbage collection, and the requirement that WebAssembly support safe interoperability with host languages (e.g., JavaScript).

Another goal of our work is to ensure that our new instructions align with with WebAssembly's performance, portability, and safety objectives. We considered the following design goals: 1) Common language features, such as green threads, should be able to compile to efficient `Wasm/k` code. 2) The extension should lend itself to simple type checking (validation). 3) The extension should lend itself to high-performance implementation in existing WebAssembly JITs. 4) The extension must be safe. 5) Existing WebAssembly instructions and code should suffer no performance penalty. And 6) the performance of new instructions should be fast and predictable.

[1] There exists a formal proposal to extend WebAssembly with exceptions [3].

Since the goal of `Wasm/k` is to provide a better compiler target language, we also prototyped `C/k`, an extension to C/C++ that adds support for delimited continuations. `C/k` uses the Emscripten compiler that compiles from C/C++ to `Wasm/k`, and we use it to implement programs with a variety of features, including green threads. We then evaluate the performance of green threads when implemented with `C/k` against the state-of-the-art approach Asyncify [28].

To summarize, we make the following contributions:

1. We design `Wasm/k`, and present its semantics, validation, and safety properties.
2. We implement `Wasm/k` as a modest extension to Wasmtime, which is a real-world WebAssembly JIT.
3. Using `Wasm/k` and the Emscripten compiler from C/C++ to WebAssembly, we present `C/k`, which adds delimited continuations to C/C++.
4. We evaluate the performance of our `Wasm/k` implementation by comparing it to a third-party tool that implements continuations by source-to-source transformation.

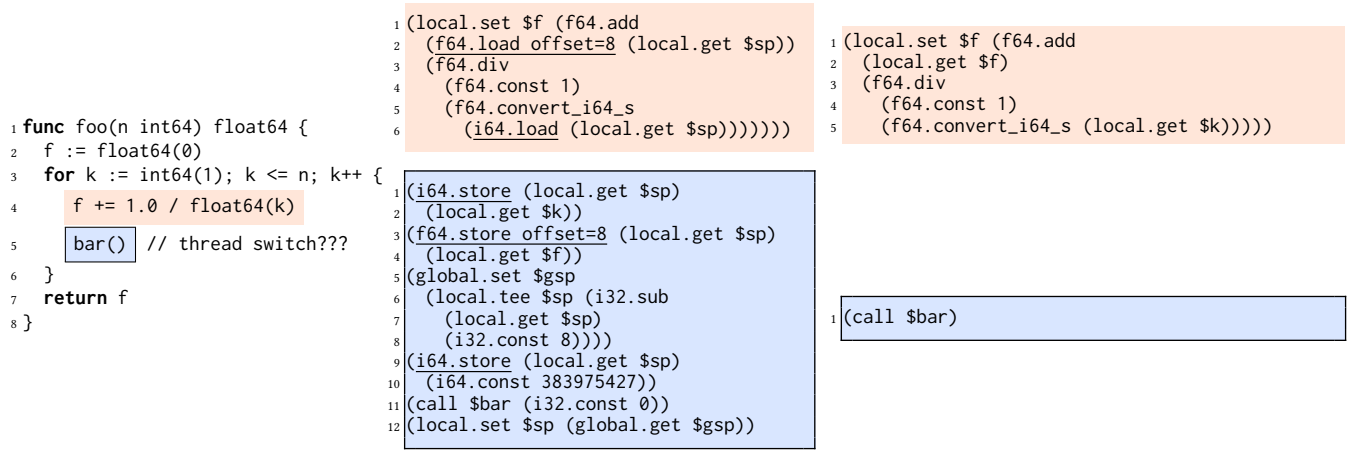
2 The `Wasm/k` Approach

We start by illustrating the basics of WebAssembly and sketch the compilation strategy that the Go compiler uses to support Goroutines in WebAssembly (Section 2.1). We then present `Wasm/k`, which extends WebAssembly with first-class continuations (Section 2.2), and `C/k`, which extends C/C++ with first-class continuations (Section 2.3). We use `C/k` to present green threads, generators, and probabilistic programming in WebAssembly (Section 2.4).

2.1 WebAssembly

WebAssembly is a stack machine with a conceptually independent control stack and value stack. For example, the (**i64.const** 2) instruction pushes the integer 2 onto the value stack, and the **i64.mul** instruction pops two integers off the stack, and pushes their product onto the stack. Similarly, functions receive their arguments and return their result on the value stack too. Each function has a collection of local variables, and can use (**local.get** $\$x$) to push a local variable's or an argument's value onto the value stack, and (**local.set** $\$x$) to pop a value off the stack and update the variable. Similarly to local variables in C, registers will be allocated for local variables during JIT compilation. For example, the `$quadruple` function (Figure 2a) quadruples its argument by calling `$helper` to first double the argument, and then doubles the result produced by `$helper`.

In addition to storing data on the stack and in local variables, data can also be stored in *linear memory* (WebAssembly's heap), which is a byte-addressable region of memory. Linear memory can be read and written via **i32.load** and **i32.store** respectively, for, e.g., the **i32** type. Unlike local



(a) Example Go program **(b)** Code generated by Go **(c)** Code generated by Emscripten for equivalent C code

Figure 1. Current WebAssembly code generation by Go and Emscripten.

variables, using linear memory will always incur hardware loads and stores.

Compiling Go to WebAssembly. *Goroutines*, which are similar to green threads (or, user-space threads), are the primary concurrency abstraction of the Go programming language. When compiled to native code, the Go runtime manages a pool of physical threads and uses them to run several Goroutines on a single thread (a so-called *M*:*N* threading model). This involves using low-level instructions to save and restore the stack and registers' values, during a (user-space) context-switch from one Goroutine to another. A context-switch may occur for a number of reasons, e.g., when the active Goroutine is blocked on I/O or simply periodically.

While physical threads are discussed in the threading proposal [8] and are available in some WebAssembly runtimes (e.g. Chrome), switching between goroutines within a single physical thread in WebAssembly is far more difficult compared to native code, since the WebAssembly stack is not allocated in linear memory and thus cannot be saved or restored. [2] Instead, the compiler generates code that maintains a heap-allocated copy of the stack residing in linear memory.

Figure 1a shows an example of a simple function `foo` in Go. The function computes a partial sum of a series, and calls the function `bar` on each iteration. Since `bar` may trigger a thread switch (Go inserts a yield at the start of every function), the compiler has to generate code to save and restore `foo`'s stack.

We sketch the generated code in Figure 1b. Before the call to `bar`, the generated code saves the local variables (`f` and `k`) onto the copy of the stack in linear memory (bottom of Figure 1b). Thus if `bar` switches to a new Goroutine, the local variables of `foo` can be safely discarded. Conversely, `foo` reads the values of its local variables from the heap-allocated stack (top of Figure 1b). Note that no such load or store instructions need to be emitted when compiling Go to native code directly, because in native code the Go runtime can freely manipulate the machine stack.

It is instructive to consider how code generation works for simpler languages, such as C. Given the C equivalent of `foo`, the Emscripten compiler from C to WebAssembly generates much simpler code (Figure 1c), without any loads and stores to linear memory. Since the code uses WebAssembly local variables exclusively, a WebAssembly JIT can easily allocate them to machine registers. (Emscripten does not support `setjmp` and `longjmp`, which can be used to build green threads in C.)

The additional loads and stores in Go have a cost. In a call to `foo`(2^{30}), the `perf` tool shows the Go program executes 2.5× more instructions, 3.0× more branches, 1.9× more loads, and 1.5× more stores than the equivalent C program, when we compile both to WebAssembly (compiled with optimizations and run with `node v14.4.0`). Overall, the Go program takes 1.8× longer than the C program. However, when compiled to native code, the performance of the C and Go code is nearly identical.

2.2 Wasm/k

Wasm/k adds five new instructions to WebAssembly. 1) The (**control** *h*) instruction captures the current continuation, stores it a region of memory called the *continuation table*, assigns it a new *continuation ID* (κ), and invokes the function (*h*) with a fresh stack, passing the continuation ID and a user-provided argument. 2) **restore** receives a continuation ID as its argument, and restores the associated continuation, discarding the current continuation in the process. It

[2] This design ensures that a malicious program cannot alter return addresses to escape the WebAssembly sandbox.

```
1 (func $helper (param $x i64) (result i64)
2    local.get $x
3    i64.const 2
4    i64.mul)
5 (func $quadruple (param $x i64) (result i64)
6    local.get $x
7    call $helper
8    i64.const 2
9    i64.mul)
```

(a) This code doubles its input first by calling a helper function, and then doubling again.

```
1 (func $handler (param $k i64) (param $x i64)
2    local.get $k
3    local.get $x
4    i64.const 2
5    i64.mul
6    restore)
7 (func $quadruple2 (param $x i64) (result i64)
8    local.get $x
9    control $handler
10   i64.const 2
11   i64.mul)
```

(b) In contrast, this code captures the current stack, and then jumps back to that stack.

Figure 2. Two ways to write a function which quadruples its input.

is a runtime error to **restore** the same continuation multiple times. 3) The **continuation_copy** instruction creates a copy of a continuation. 4) The **continuation_delete** instruction deletes a continuation without restoring it. 5) The **prompt** e^* **end** instruction wraps a block of instructions (e^*), and serves as a delimiter for continuation capture: all continuations captured by e^* do not extend beyond the call to prompt.

Figure 2b shows another implementation of `$quadruple`, using continuations in a trivial way. The function captures its continuation and passes it to `$handler` (line 9), which runs in the empty continuation. The `$handler` function receives two arguments: `$k` is the captured continuation ID, and `$x` is the original argument to `$quadruple2`, passed through to `$handler`. The `$handler` function doubles its argument and restores `$k`, passing the doubled value along. At this point, the captured continuation will execute (line 10), with the doubled valued pushed onto the stack. Execution completes as before by doubling again.

2.3 C/k: Continuations for C/C++

Writing and reading substantial examples in WebAssembly is tedious. Therefore, the rest of this paper presents examples using C/C++. We use the Emscripten compiler from C/C++ to WebAssembly, and export the new `Wasm/k` instructions to C/C++ programs using the API defined in Figure 3. Each of these functions call their corresponding instructions in `Wasm/k` to manipulate the WebAssembly stack.

```
1 typedef uint64_t k_id;
2 typedef void (*control_handler_fn)(k_id, uint64_t);
3
4 uint64_t control(uint64_t arg, control_handler_fn fn_ptr);
5 void restore(k_id k, uint64_t val);
6 uint64_t continuation_copy(k_id k);
7 void continuation_delete(k_id k);
8 #define prompt(x) <...>
```

Figure 3. A C/C++ First-Class Continuations Header

However, it is not enough to directly expose the `Wasm/k` primitives to C++ code. A program written in C/C++ can get the memory address of a local variable, which WebAssembly does not support. Emscripten uses a heap-allocated portion of the stack to support these programs. Therefore, `C/k` has to carefully manage this portion of the stack as well (Section 4).

Adding first-class continuations in this manner to C/C++ is unusual, as typically first-class continuations are a feature in high-level languages such as Racket, and need to be compiled to low-level code which does not support first-class continuations. By going in the opposite direction, we get them almost for free in a higher-level language.

2.4 Using Continuations in Wasm/k and C/k

We now present several applications of `Wasm/k`, using `C/k` to write our code.

Green threads. Green threads (or cooperative threads), are a simple example of an abstraction that is easy to build with continuations. Figure 4b shows how to use `C/k` to build an implementation of green threads, which provides functions to create new threads, wait on threads to complete, and suspend the running thread and yield control to another thread (`thread_yield`). Figure 4a is a small program that uses this threading library.

The key insight is that `thread_yield` can be accomplished by capturing the current continuation of the thread via `control` (Figure 4b line 37), storing the continuation ID in a queue (Figure 4b line 33), and dequeue-ing and restoring another continuation ID (Figure 4b line 34). Since green threads do not need to pass data between threads, we do not utilize the data arguments to `control` and `restore`.

Generators. *Generators* are a programming abstraction that are found in a variety of languages, including Python and JavaScript. Although C does not support generators, we can build them using `control` and `restore`. Figure 5a shows a program in `C/k` that prints the numbers 0 through 9, using a generator function. The generator contains what appears to be an infinite loop, but each iteration suspends execution in the generator (`gen_yield`) and resumes execution in `main`.

Figure 5b presents the implementation of generators using `C/k`. The primary difference between our implementation and canonical implementations (e.g., in Racket [4]), is that C

```
void thread_main() {
    std::cout << "A" << std::endl;
    thread_yield();
    std::cout << "B" << std::endl;
}
int main() {
    thread_create(thread_main);
    thread_create(thread_main);
    join_all_threads();
}
```

(a) Example of use. Prints AABB

```
std::vector<uint64_t> Q;
uint64_t after_join;
uint64_t dequeue() {
    uint64_t next_k = Q.back(); Q.pop_back();
    return next_k;
}

void save_fk_restore(uint64_t fk, uint64_t create_k) {
    restore(create_k, fk);
}
void create_handler(uint64_t k, uint64_t f) {
    control(save_fk_restore, k);
    ((void (*)())f)();
    if(Q.size() > 0) {
        restore(dequeue(), 0);
    } else {
        restore(after_join, 0);
    }
}
void thread_create(void (*f)()) {
    Q.insert(Q.begin(), control(create_handler, (uint64_t)f));
}

void join_handler(uint64_t k, uint64_t arg) {
    after_join = k;
    restore(dequeue(), 0);
}
void join_all_threads() {
    control(join_handler, 0);
}

void yield_handler(uint64_t k, uint64_t arg) {
    Q.insert(Q.begin(), k);
    restore(dequeue(), 0);
}
void thread_yield() {
    control(yield_handler, 0);
}
```

(b) Implementation.

Figure 4. Green threads in C/k

```
void example_generator(Generator *g) {
    uint64_t i = 0;
    while(1) { gen_yield(i++, g); }
}
int main() {
    Generator *g = make_generator(example_generator);
    for(int i = 0; i < 10; i++)
        printf("%llu\n", gen_next(g));
    free_generator(g);
    return 0;
}
```

(a) Example of use.

```
typedef struct {
    k_id after_next, after_yield; uint64_t value;
} Generator;
// Helpers for converting a function to a continuation
void return_convert_result(uint64_t k, uint64_t ak) {
    restore(ak, k);
}
void convert_handler(uint64_t k, void (*f)(Generator*)) {
    f((Generator *)control(return_convert_result, k));
}
uint64_t convertFuncToCont(void (*f)(Generator*)) {
    return control(convert_handler, f);
}
// Allocating a generator
Generator *make_generator(void (*f)(Generator*)) {
    Generator *g = (Generator *)malloc(sizeof(Generator));
    g->after_yield = convertFuncToCont(f); return g;
}
// Yielding implementation
void yield_handler(k_id k, Generator *g) {
    g->after_yield = k;
    restore(g->after_next, g->value);
}
void gen_yield(uint64_t v, Generator *g) {
    g->value = v;
    control(yield_handler, g);
}
// Next implementation
void next_handler(k_id k, Generator *g) {
    g->after_next = k;
    restore(g->after_yield, 0);
}
uint64_t gen_next(Generator *g) {
    return control(next_handler, g);
}
// Freeing a generator
void free_generator(Generator *g) {
    continuation_delete(g->after_yield); free(g);
}
```

(b) Implementation.

Figure 5. Generators in C/k.

does not support first-class functions. Therefore, we represent a generator as an object (`struct`) with fields that hold 1) the ID of the continuation where the generator was invoked (`after_next`), 2) the ID of the continuation where the generator was last suspended (`after_yield`), and 3) the next value to return from the generator (`value`).

Finally, the generator API includes a function to delete a generator object (`free_generator`). This function deletes the continuation within the generator (`g->after_yield`) using `continuation_delete` (line 38). Note that since the other continuation (`g->after_next`) was restored to during the most recent yield (line 22), it is currently unallocated and does not need to be deleted. The necessity of a **continuation_delete** instruction is subtle, but is required for natural use cases of first-class continuations in a low-level language without garbage collection.

```
uint64_t sum_d6() {
    auto *d6 = new std::vector<uint64_t> {1, 2, 3, 4, 5, 6};
    return uniform(d6) + uniform(d6);
}
int main() {
    std::cout << *driver(sum_d6) << std::endl; return 0;
}
```

(a) Example of use.

```
1  struct ContinuationThunk {
2      k_id continuation; // The continuation to resume
3      uint64_t value; // The value to pass to the continuation
4  };
5  // vector of thunks which need to be executed
6  std::vector<ContinuationThunk *> to_execute;
7
8  std::map<uint64_t, double> *driver(uint64_t (*body)()) {
9      auto *results = new std::vector<uint64_t>();
10     results->push_back(body());
11     if(rest.size() > 0) {
12         ContinuationThunk *t = rest.back(); rest.pop_back();
13         restore(t->continuation, t->value);
14     }
15     return count_probs(results);
16 }
17
18 void uniform_handler(k_id k, std::vector<uint64_t> *args) {
19     for(auto it = std::next(args->begin());
20         it != args->end(); ++it) {
21         to_execute.push_back(new ContinuationThunk {
22             .continuation=continuation_copy(k),
23             .value=*it});
24     }
25     restore(k, args[0]);
26 }
27 uint64_t uniform(std::vector<uint64_t> *args) {
28     return control(uniform_handler, args);
29 }
```

(b) Implementation.

Figure 6. An embedded probabilistic programming language in C++.

Probabilistic programming. A more involved example is the implementation of an embedded probabilistic programming language in C++. Probabilistic programming languages allow probabilistic models to be implemented declaratively in general purpose languages. One common approach to implement a probabilistic programming language is to relate sampling from a probability distribution to sampling from a distribution of program executions [21]. Performing this sampling requires some use of control operators which can essentially fork execution to allow it to be re-executed (i.e., sampled from) multiple times. While the implementation of a proper probabilistic programming language with modern sampling algorithms is out of the scope of this paper, we can nevertheless demonstrate how to implement a probabilistic programming language allowing for finite distributions embedded in C++.

An example usage of the embedded probabilistic programming language is shown in Figure 6a. `sum_d6` computes the sum of two independent dice rolls. The call to `driver(sum_d6)` will run the sampling algorithm, eventually returning a map that represents the probability mass function (PMF) of `sum_d6`. The proposed API consists of just a `uniform` function which represents the uniform distribution over a discrete set of values (the vector argument) and the `driver` function which conducts the sampling to obtain the final PMF. This API can be easily expanded in this framework to allow for different distributions and conditioning, but these are omitted for brevity.

The implementation of the API is shown in Figure 6b. The core idea is that each sample from a distribution will correspond to forking the execution for each sampled value. For example, if sampling from `uniform(1, 2, 3)` the execution would be forked into 3 executions, one with each sampled value. The various execution forks are stored in the `to_execute` state, in the format of a vector of `ContinuationThunks` (lines 1–6), which keep track of the continuation to restore to, and the sampled value to pass to the continuation upon restoring.

The implementation of `driver` (lines 8–16) keeps a vector of final sampled values, and proceeds by first running the given function argument (`body()`) and saving the result, and then dequeuing a thunk to execute and restoring it. Supposing that `body` forked its execution into thunks, then the call to `restore` (line 13) will jump back into the execution of somewhere in `body`, eventually returning yet again to the `push_back` (line 10). Thus, `driver` will continue to push results and dequeue a new thunk, until all thunks (samples) are exhausted. Finally, `count_probs` computes the desired map.

With `driver` worked out, the implementation of `uniform` is conceptually straightforward: `uniform(args)` should fork the execution for each value in `args`. This is accomplished by first immediately calling `control` (line 28) to capture the current continuation. Then, for every element except the first element of `args` a new thunk is queued, where the continuation is a *copy* of the current continuation `k` (line 22). An explicit copy of `k` is required because all of these thunks will eventually be restored to, and under one-shot continuation semantics it is invalid to restore to a single continuation (`k`) multiple times. Finally, the current continuation is restored immediately with the first sampled value rather than saved in a thunk (line 25).

3 Semantics of `Wasm/k`

This section presents 1) an overview of WebAssembly's operational semantics, 2) extends the operational semantics to support continuations, 3) presents type-checking (known as validation) for this extension, and 4) proves that the extension is sound.

```
(i64.const 2)     ;; push 2
block             ;; enter block
(i64.const 3)     ;; push 3
(i64.const 4)     ;; push 4
i64.add           ;; pop 3 & 4, push 7
(br 0)            ;; exit block
end               ;; end of block
i64.sub           ;; pop 2 & 7, push −5
```

(a) Example program.

(i64.const 2) block (i64.const 3) (i64.const 4) i64.add (br 0) end i64.sub
↪ (i64.const 2) label{ϵ} (i64.const 3) (i64.const 4) i64.add (br 0) end i64.sub
(within an L^1 context)
↪ (i64.const 2) label{ϵ} (i64.const 7) (br 0) end i64.sub
↪ (i64.const 2) (i64.const 7) i64.sub
↪ (i64.const (−5))

(b) Reduction sequence.

Figure 7. An example of WebAssembly execution.

3.1 WebAssembly Semantics

WebAssembly is formalized as a stack-based, small-step reduction semantics. This section introduces a small fragment of the WebAssembly semantics, using the example program in Figure 7a. For a more detailed account, we refer the reader to Haas et al. [18] and the WebAssembly specification [11].

The WebAssembly stack machine contains both instructions (e) that are pending evaluation and values (v) that were produced by instructions that have already been evaluated. The values are a subset of instructions. For example, the instruction (**i64.const** n) pushes the 64-bit value n onto the stack. Moreover, the semantics represents the 64-bit value n as (**i64.const** n). In the absence of control flow and function calls, a configuration of the WebAssembly stack machine has a sequence of evaluated values (v^*) and a sequence of instructions (e^*) in succession (v^*e^*), and we always evaluate the first non-value instruction in the sequence. The boundary between values and instructions at which evaluation occurs is called the *local context of depth 0* ($L^0[\_]$).

WebAssembly has *structured control flow*, and does not have goto-style instructions. Instead, the language has structured control flow blocks (e.g., **block** … **end** and **loop** … **end**). The WebAssembly semantics turns all kinds of blocks into labelled blocks (**label**$\{e^*\}$ … **end**), which are an administrative instruction.[3] The nested structure of labelled blocks is defined by *local contexts of depth k* ($L^k[\_]$). A local context of depth 0 ($L^0[\_]$) matches a stack of the form v^*e^*, and a local context of depth $k+1$ matches a local context of depth k nested inside a labelled block. For example, Figure 7a has four instructions within an L^1 context.

A WebAssembly program is organized as a collection of *modules* that import and export code and data. An instantiated module with no unresolved imports is called an *instance*. The global execution state of all instances is called the *store*. We present Wasm/k as an extension to the WebAssembly formal semantics, which includes the machinery needed to support multiple instances. However, for the purpose of this paper, it is sufficient to consider programs with just one instance.

[3]The e^* is only needed to encode loops, and can be ignored in this paper.

Continuation IDs
κ_id ::= i64

Instructions
e ::= ⋯
| **control** h
| **restore**
| **continuation_copy**
| **continuation_delete**
| **prompt** $tf\ e^*$ **end**

Full-Stack Contexts
$L^{\max}$ ::= $v^*[\_]e^*$ | v^* **label**$_n\{e^*\}$ $L^{\max}$ **end** e^*

Stores
s ::= {inst $inst^*$, ⋯ }

Instances
$inst$ ::= {func cl^*, glob v^*, tab $i^?$, mem $i^?$, pstack $pstack$ }

Continuation Table Stacks
$pstack$ ::= $pinst^*$

Continuation Tables
$pinst$ ::= {ctable ({locals v^*, ctx $L^{\max}$, inst i} | nil)*, root (κ_id | nil)}

Figure 8. Syntax of Wasm/k: we extend the WebAssembly runtime structure with a table of continuations.

The WebAssembly stack, nested control flow, the store, and instances are the elements of WebAssembly that are relevant to Wasm/k. With these defined, WebAssembly has a small-step semantics that updates the stack, and possibly the store (s) and local variables (v_l^*) at each step ($s; v_l^*; e^* \hookrightarrow s'; v_l'^*; e'^*$). The semantics is congruent with local contexts: if $s; v_l^*; e^* \hookrightarrow s'; v_l'^*; e'^*$ then $s; v_l^*; L^k[e^*] \hookrightarrow s'; v_l'^*; L^k[e'^*]$. Thus evaluation always occurs in the innermost local context, unless no such evaluation is possible. When an instruction does not read or write from the store, we omit the store for brevity. Figure 7b shows the execution trace of our example.

3.2 Design and Semantics of Wasm/k

We first describe the new values and types of Wasm/k, then present necessary changes to WebAssembly instances, and finally present the new reduction rules of Wasm/k.

The continuation table and continuation IDs. In a language that supports first-class continuations, a continuation

is a new kind of value. First-class continuations are typically found in high-level languages (e.g., Scheme or Racket) that also support first-class functions. This allows functions that receive captured continuations (e.g., the argument to `call/cc` in Scheme) to close over other variables in their environment. However, this is not possible in WebAssembly, since it lacks first-class functions. Moreover, it is not straightforward to safely add new kinds of values to WebAssembly either. (The WebAssembly heap is untyped and byte-addressable, so a program can make arbitrary changes to the representation of any value stored on the heap.) `Wasm/k` adds a *continuation table*, which associates a continuation with an integer-valued *continuation ID* (***κ_id***). The `Wasm/k` runtime system manages the table, and `Wasm/k` programs work with continuations indirectly by referring to their ID. Since continuation IDs are standard integers, programs can use existing load and store instructions to save continuation IDs in linear memory. However, the new `Wasm/k` instructions have to dynamically ensure that they receive valid continuation IDs. Our implementation uses 64-bit integers to represent continuation IDs.

`Wasm/k` has delimited continuations, which are needed to safely interoperate with host languages such as JavaScript (Section 3.4). Thus `Wasm/k` includes a **prompt** instruction, and we modify instances to track a stack of dynamic **prompt** scopes. Formally, we extend instances as follows (highlighted in Figure 8). Each instance (*inst*) contains a stack of dynamically nested **prompt** contexts. Each prompt context (*pinst*) is a record containing a continuation table and a continuation ID of the *root continuation*, defined to be the continuation associated with the stack which initialized the WebAssembly execution or most recent prompt. The continuation table consists of an array of entries, where each entry is either *nil* or a captured continuation, with all entries initialized to *nil*.[4]

A continuation saved in a continuation table is a record that contains the values of local variables (across all stack frames), and the entire evaluation context at the point of capture. However, WebAssembly's local context (L^k) only capture a stack with k nested blocks. Therefore, we define *full-stack contexts* ($L^{\max}$) as evaluation contexts that match a stack with arbitrary control block depth, and a continuation stores a full-stack context.

Finally, the root continuation ID in a continuation table is maintained such that if the root continuation is currently executing, *root* will be set to *nil*, otherwise *root* will be set to the index κ_R of the root continuation in the continuation table.

[4]To control resource utilization, WebAssembly implementations can define the maximum size of various dynamic and static data structures, e.g., the number of stack frames. Similarly, we impose an implementation-dependent bound on the number of allocated continuations.

New reduction rules. The semantics of WebAssembly define a reduction relation ($\hookrightarrow$) that is congruent with local contexts ([Cong] in Figure 9). However, full congruence with local contexts does not hold in the presence of first-class continuations. Therefore, `Wasm/k` introduces a new reduction relation ($\rightsquigarrow$) for programs that contain (**control** h) and **restore** instructions (Figure 9). The extended semantics refer to the original WebAssembly reduction relation ($\hookrightarrow$), using the [Cong] rule, but there is no equivalent rule for $\rightsquigarrow$. If there is a reduction which involves no use of (**control** h) or **restore**, then it is also a valid reduction which *might* make use of (**control** h) or **restore**, as given in the [No-Ctrl] rule.

The (**control** h) instruction receives a single argument (v) and calls the function h, passing it a new continuation ID (κ) and the argument v. The continuation ID is bound to the current continuation ($L^{\max}$) and local variables (v_l^*), and the call to h is followed by a **trap**: i.e., it is a runtime error to return normally from h. For simplicity, (**control** h) makes a direct call to a function h. However, when an indirect call is necessary, it is possible to use v to pass the index of a function to h.

The **restore** instruction receives a continuation ID (κ) and a restore value (v). The instruction dynamically checks that κ is a valid continuation ID. If κ is valid, it restores the local variables ($v_l^{*\prime}$) and the stack ($L^{\max\prime}$) that is associated with κ, and returns v to the stack. The **restore** instruction also marks the continuation ID (κ) as *nil* in the continuation table, which allows it to be reused by subsequent calls to (**control** h). Finally, when restoring the root continuation, **restore** sets the root ID back to *nil*, and leaves it untouched otherwise. Note that **restore** is abortive rather than functional, in the sense that **restore** aborts the current continuation and instructions following **restore** will never be executed. It is a runtime error to call **restore** on a continuation ID (κ) that is un-allocated, or to invoke **restore** within the root continuation. In either case, a **trap** occurs.

We need the **continuation_copy** instruction to create a copy of a saved continuation, so that a program can restore a continuation several times if needed. This instruction assigns a new continuation ID to the copy. A **trap** occurs if the provided continuation ID is mapped to *nil*. The **continuation_delete** instruction deallocates an continuation without restoring it, and may be needed to avoid memory leaks in certain applications.

In the presence of first-class continuations, a function f may now never return to the call site or may return multiple times. Motivated by a need for safe FFI, the goal of a **prompt** *tf* e^* **end** instruction [5] is to evaluate the body e^* such that e^* is guaranteed to finish evaluation exactly once (or trap/diverge), and trap otherwise. Note that this is similar to Felleisen's prompt [15], but in cases where Felleisen's

[5]*tf* is a type annotation of the body (e^*) of the prompt, and is not important to understand the semantics.

$$\boxed{s;\ v^*;\ e^* \rightsquigarrow_i s;\ v^*;\ e^*}$$

$$\text{[Cong]}\ \frac{s;\ v^*;\ e^* \hookrightarrow_i s';\ v'^*;\ e'^*}{s;\ v^*;\ L^k[e^*] \hookrightarrow_i s';\ v'^*;\ L^k[e'^*]} \qquad \text{[No-Ctrl]}\ \frac{s;\ v^*;\ e^* \hookrightarrow_i s';\ v'^*;\ e'^*}{s;\ v^*;\ e^* \rightsquigarrow_i s';\ v'^*;\ e'^*}$$

$$\begin{array}{lll}
\text{[Ctrl]} & s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ v)\ (\textbf{control}\ h)] \rightsquigarrow_i s';\ \epsilon;\ (\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ (\textbf{call}\ h)\ \textbf{trap} & \text{if } (s', \kappa) = \delta_{\text{ctrl}}(s, i, v_l^*, L^{\max}) \\
\text{[Restore]} & s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\textbf{i64.const}\ v)] & \text{if } (s', v_l^*, L^{\max\prime}) = \delta_{\text{rest}}(s, i, \kappa) \\
\text{[Restore-Err]} & s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i s;\ v_l^*;\ \textbf{trap} & \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
\text{[Copy]} & s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation_copy} \hookrightarrow_i s';\ (\textbf{i64.const}\ \kappa') & \text{if } (s', \kappa') = \delta_{\text{copy}}(s, i, \kappa) \\
\text{[Copy-Err]} & s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation_copy} \hookrightarrow_i s;\ \textbf{trap} & \text{otherwise} \\
\text{[Delete]} & s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation_delete} \hookrightarrow_i s';\ \epsilon & \text{if } s' = \delta_{\text{delete}}(s, i, \kappa) \\
\text{[Delete-Err]} & s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation_delete} \hookrightarrow_i s;\ \textbf{trap} & \text{otherwise} \\
\text{[Prompt]} & s;\ \textbf{prompt}\ tf\ e^*\ \textbf{end} \hookrightarrow_i s';\ \textbf{block}\ tf\ e^*\ \textbf{end}\ \textbf{prompt_end} & \text{if } s' = \delta_{\text{p}}(s, i) \\
\text{[Prompt-End]} & s;\ \textbf{prompt_end} \hookrightarrow_i s';\ \epsilon & \text{if } s' = \delta_{\text{p-end}}(s, i)
\end{array}$$

$$\delta_{\text{ctrl}}(s, i, v_l^*, L^{\max}) ::= \begin{cases} (\text{setCont}(\text{setRoot}(s, i, \kappa), i, \kappa, \{\text{locals} = v_l^*, \text{ctx} = L^{\max}, \text{inst} = i\}),\ \kappa) & \text{if } \text{getRoot}(s, i) = \mathit{nil} \\ (\text{setCont}(s, i, \kappa, \{\text{locals} = v_l^*, \text{ctx} = L^{\max}, \text{inst} = i\}),\ \kappa) & \text{if } \text{getRoot}(s, i) \neq \mathit{nil} \end{cases}$$
where κ is fresh, i.e., $\text{getCont}(s, i, \kappa) = \mathit{nil}$

$$\delta_{\text{rest}}(s, i, \kappa) ::= \begin{cases} (\text{setRoot}(\text{setCont}(s, i, \kappa, \mathit{nil}), i, \mathit{nil}),\ \text{getCont}(s, i, \kappa)_{\text{locals}},\ \text{getCont}(s, i, \kappa)_{\text{ctx}}) & \text{if } \text{getRoot}(s, i) = \kappa \\ (\text{setCont}(s, i, \kappa, \mathit{nil}),\ \text{getCont}(s, i, \kappa)_{\text{locals}},\ \text{getCont}(s, i, \kappa)_{\text{ctx}}) & \text{if } \mathit{nil} \neq \text{getRoot}(s, i) \neq \kappa \end{cases}$$

$$\delta_{\text{copy}}(s, i, \kappa) ::= (\text{setCont}(s, i, \kappa', \text{getCont}(s, i, \kappa)),\ \kappa') \quad \text{if } \text{getRoot}(s, i) \neq \kappa \wedge \text{getCont}(s, i, \kappa) \neq \mathit{nil}$$
where κ' is fresh, i.e., $\text{getCont}(s, i, \kappa') = \mathit{nil}$

$$\delta_{\text{delete}}(s, i, \kappa) ::= \text{setCont}(s, i, \kappa, \mathit{nil}) \quad \text{if } \text{getRoot}(s, i) \neq \kappa \wedge \text{getCont}(s, i, \kappa) \neq \mathit{nil}$$

$$\delta_{\text{p}}(s, i) ::= s' \text{ where } s' = s \text{ except } s'_{\text{inst}}(i)_{\text{pstack}} \mapsto \text{push}(s_{\text{inst}}(i)_{\text{pstack}}, \{\text{ctable} = \mathit{nil}^*, \text{root} = \mathit{nil}, \text{inst} = i\})$$

$$\delta_{\text{p-end}}(s, i) ::= s' \text{ where } s' = s \text{ except } s'_{\text{inst}}(i)_{\text{pstack}} \mapsto \text{pop}(s_{\text{inst}}(i)_{\text{pstack}}) \quad \text{if } \text{getRoot}(s, i) = \mathit{nil}$$

$$\begin{aligned}
\text{getRoot}(s, i) &::= \text{top}(s_{\text{inst}}(i)_{\text{pstack}})_{\text{root}} \\
\text{getCont}(s, i, \kappa) &::= \text{top}(s_{\text{inst}}(i)_{\text{pstack}})_{\text{ctable}}(\kappa) \\
\text{setRoot}(s, i, \kappa_R^?) &::= s' \text{ where } s' = s \text{ except } \text{top}(s'_{\text{inst}}(i)_{\text{pstack}})_{\text{root}} \mapsto \kappa_R^? \\
\text{setCont}(s, i, \kappa, \gamma^?) &::= s' \text{ where } s' = s \text{ except } \text{top}(s'_{\text{inst}}(i)_{\text{pstack}})_{\text{ctable}}(\kappa) \mapsto \gamma^?
\end{aligned}$$

Figure 9. Semantics of Wasm/k.

prompt alters the control flow, Wasm/k's **prompt** traps. This design is due to the fact that our **restore** operator is abortive rather than functional. Evaluation of **prompt** *tf* e^* **end** involves first pushing a prompt context onto the prompt stack with a blank continuation table and the *root* ID set to *nil*, then executing e^* inside a scoped block, and finally executing the administrative non-user accessible instruction **prompt_end**. Note that if e^* were to contain branches to labels outside of the **prompt**, the execution of **prompt_end** could be skipped. The validation rules discussed below outlaw such branches. The safety properties of **prompt** during FFI is discussed in Section 3.4. Evaluating a **prompt_end** instruction pops and discards the top prompt context from the prompt stack.

3.3 Validation

Validation (type checking) is accomplished in WebAssembly by assigning each instruction a type describing the values it pops from the stack and the values it pushes onto the stack. For example, the type of an add instruction (**i32.add**) is i64 i64 → i64. In addition, the context (C) stores information during the type checking algorithm, such as the types of functions.

Figure 10 shows the type checking rules for Wasm/k. The type checking of Wasm/k fits easily into the existing type checking framework of WebAssembly, since we check dynamically that continuation IDs are valid, similar to the type checking of indirect function calls.

The type checking of **restore**, **continuation_copy**, and **continuation_delete** instructions is straightforward as they are all typed independent of the context (C). In particular, these instructions do not statically type check validity of continuation IDs, beyond being the correct type (i64), since the semantics in Figure 9 check continuation ID validity at runtime. The type checking of a (**control** h) instruction does

$$C ::= \{\ldots, \text{label}\ ((t^*)^*)^*, \text{pstack}\{\text{ctable}(t^* \mid nil)^*, \text{root}(\kappa_R \mid nil)\}^*\}$$

$$\frac{C_{\text{func}}(h) = \text{i64 i64} \rightarrow \epsilon}{C \vdash (\textbf{control}\ h) : \text{i64} \rightarrow \text{i64}}$$

$$\frac{}{C \vdash \textbf{restore} : t_1^*\ \text{i64 i64} \rightarrow t_2^*}$$

$$\frac{}{C \vdash \textbf{continuation\_copy} : \text{i64} \rightarrow \text{i64}}$$

$$\frac{}{C \vdash \textbf{continuation\_delete} : \text{i64} \rightarrow \epsilon}$$

$$\frac{tf = t_1^n \rightarrow t_2^m \qquad C\{\text{label} = C_{\text{label}}; ((t_2^m)),\ \text{return} = \epsilon\} \vdash e^* : tf}{C \vdash \textbf{prompt}\ tf\ e^*\ \textbf{end} : tf}$$

Figure 10. Type Checking of Wasm/k.

$$\text{[Root]}\quad \frac{\vdash_i s;\ v^*;\ e^* : t^* \qquad \vdash s : S \qquad S_{\text{inst}}(i)_{\text{roots}} = nil^*}{\vdash_i^k s;\ v^*;\ e^* : t^*}$$

$$\text{[Non-Root]}\quad \frac{\begin{array}{cc}\vdash s : S & p_R = \max\{p \mid S_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} \neq nil\} \\ \vdash_i s;\ v^*;\ e^* : t^* & \kappa_R = S_{\text{inst}}(i)_{\text{pstack}}(p_R)_{\text{root}}\end{array}}{\vdash_i^k s;\ v^*;\ e^* : S_{\text{inst}}(i)_{\text{pstack}}(p_R)_{\text{ctable}}(\kappa_R)}$$

Figure 11. Typing delimited instructions.

involve checking a side condition in the context: in order to type check (**control** h), the handler function (h) is looked up in the context (C), and checked to have the correct type of a control handler function (receives two i64 arguments and returns nothing).

Type checking the **prompt** instruction is the most interesting case. Semantically, **prompt** $tf\ e^*$ **end** must 1) prepare the prompt environment, 2) execute e^*, and 3) teardown the prompt environment (i.e., execute the **prompt_end** administrative instruction). However, consider that e^* may contain branch instructions jumping to labels lexically outside of the **prompt**, which would then incorrectly be able to jump beyond tearing-down of the prompt environment. To remedy this, we use the type checker to outlaw branching instructions which jump beyond the scope of the **prompt**, though still allow branches within the **prompt**.

We extend type-checking contexts (C) to store a stack of stacks of labels, as shown in the top of Figure 10 ($\text{label}((t^*)^*)^*$). Implicitly, we define the notation of context label extension $C, \text{label}(t^*)$ used in previous WebAssembly type checking rules to mean that the label (t^*) is pushed onto the *top-most* stack in C (or in a new stack if none exist), and likewise the notation $C_{\text{label}(i)}$ we define to mean indexing by i into the *top-most* stack in C. These implicit re-definitions allow all the other WebAssembly type checking rules to remain untouched. With this machinery in place, the type checking rule for **prompt** can be given, which closely mirrors the type checking rule of **block**, except that an entire new label stack is pushed into the context and the return label is invalidated.

An alternative approach could be to modify the semantics to force the **prompt_end** instruction to be run even when branching past it. However, this would require significant changes to how branch instructions are specified in the WebAssembly semantics, and would significantly impact code generation.

3.4 Safety Properties of Wasm/k

We first prove the safety of Wasm/k, building on the safety of WebAssembly. We then consider safe interoperation with a host language.

***Safety of standalone* Wasm/k.** WebAssembly is equipped with a syntactic type soundness theorem [18, 19, 26], which we build on.

WebAssembly's instruction typing relation ($\vdash_i e^*; t^*$) calculates a sequence of types (t^*), which specify the types of the values that are left on the stack by the instructions (e^*). These types are preserved by each step of evaluation ($\hookrightarrow$). However, if a step captures or restores a continuation ($\rightsquigarrow$), the type of the current instruction sequence may change.

To address this, we introduce a new typing relation ($\vdash_i^k$) which extracts the type of the unique stack nested most deeply in prompts which has been invoked through a chain of root stacks (Figure 11). We call this stack the *primary root stack*. There are two cases to this relation: 1) when the current instruction sequence is the primary root stack, we return its type ([Root]), and 2) if not, we extract the type of the saved primary root stack from the store ([Non-Root]). Using this typing relation, we prove progress and preservation for Wasm/k.

Theorem 3.1 ($\rightsquigarrow$ Preservation)**.** *If* $\vdash_i^k s;\ v^*;\ e^* : t^*$ *and* $s;\ v^*;\ e^* \rightsquigarrow_i s';\ v'^*;\ e'^*$, *then* $\vdash_i^k s';\ v'^*;\ e'^* : t^*$.

Theorem 3.2 ($\rightsquigarrow$ Progress)**.** *If* $\vdash_i^k s; v^*; e^* : t^*$, *then either* $e^* = v'^*$ *or* $e^* = \textbf{\textit{trap}}$ *or* $s; v^*; e^* \rightsquigarrow_i s'; v'^*; e'^*$.

Both proofs are available in the supplemental appendix [23].

Safe interoperation. A WebAssembly runtime environment is typically embedded in a host language, and offers an API that allows function calls from either language to the other. For example, Wasmtime supports interoperability with Rust, and web browsers support interoperability with JavaScript. Neither Rust nor JavaScript support continuations, and require foreign function calls to return exactly once. The **prompt** operator allows us to enforce this dynamically. Wasm/k automatically inserts a **prompt** block around a foreign call into Wasm/k. This design is similar to Scheme / Racket, but differs in two regards. First, Scheme / Racket allow FFI to be unsafe as they do not forcibly wrap every FFI call in a prompt, while Wasm/k prioritizes safety over some

flexibility. Second, Scheme / Racket will not abort the program upon control flow which violates the exactly-once semantics of `prompt`, but will instead alter the control flow [15]. In keeping with using **prompt** strictly to enforce FFI safety, `Wasm/k` considers it a programmer error to attempt to violate such safety.

4 Leveraging `Wasm/k` in Existing Compilers

Since `Wasm/k` does not alter the semantics of existing WebAssembly instructions, it ought to be easy to use `Wasm/k` to implement continuations in an existing compiler. However, today's compilers use code generation techniques that require a little extra care.

For example, consider Emscripten, which compiles C to WebAssembly. A typical C compiler would allocate local variables on the machine stack, and Emscripten is no exception. However, whereas a C program can obtain a pointer to a local, stack-allocated variable—a common operation in C programs—it is not possible to do so in WebAssembly. The WebAssembly stack is not stored in linear memory, and programs can only obtain pointers to values in linear memory. Therefore, to support these programs, Emscripten allocates local variables on the WebAssembly stack when possible, but uses linear memory when necessary. Emscripten generates code that reserves a block of memory to store the heap-allocated portion of the stack, and uses global variables that emulate stack and frame pointers.

Section 2.3 presented `C/k`, which extends Emscripten with continuations. Our extension adds new library functions that each correspond to a `Wasm/k` instruction, which manage saving and restoring the WebAssembly stack. However, we need to ensure that these operations correctly save and restore the heap-allocated portion of the stack (`Wasm/k` cannot do this automatically, since it is source-language neutral). Therefore, we insert code at the call site for each `C/k` operation to manipulate Emscripten's global stack pointer values. For example, at a call site of `control`, we insert code that saves the current heap-allocated portion of the stack and the value of the stack pointer into a table. Similarly, at a call site of `restore`, we insert code that restores the heap-allocated portion of the stack and stack pointer from the table.

This problem is not unique to Emscripten. For example, the Go compiler's WebAssembly backend creates a copy of the WebAssembly stack in linear memory to support garbage collection and Goroutines. We speculate that `Wasm/k` would allow the Go compiler to store non-pointer variables on the WebAssembly stack, which may improve the performance of numeric code. However, GC roots would still have to be stored in linear memory.

5 Implementation

We implement `Wasm/k` as an extension to *Wasmtime*, which is a standalone, JIT-based runtime system for WebAssembly.[6] Wasmtime is written in Rust and primarily developed by Mozilla. Wasmtime, and other WebAssembly JITs, use the native machine stack to store both values and return addresses. Wasmtime also performs register allocation to avoid using the stack when possible. Therefore, our `Wasm/k` implementation has to manage the native stack and registers, and take care to follow the calling convention that Wasmtime employs.

Capture and restore. The implementation of (**control** h) involves several steps. 1) It uses a free list to allocate an unused continuation ID. 2) It associates this continuation ID with a new continuation object, which holds the values of machine registers that are not caller-saved, which includes the stack and instruction pointers. 3) It allocates a new block of memory to hold subsequent stack frames, and sets the stack pointer to point to this block of memory. 4) It jumps to the WebAssembly function h, which receives the new continuation ID. To further improve performance, we preallocate a pool of memory to hold new stacks.

The implementation of **restore** is straightforward, since its principal task is to restore the registers saved by **control** in the continuation object. To ensure safety, we 1) ensure that the continuation ID is associated with a valid continuation object, 2) delete the continuation object so that it cannot be restored again, and 3) reclaim the memory used by the current stack.

Copying continuations. To copy a continuation, we allocate a new continuation ID, and duplicate a continuation object, but have to carefully tackle all pointers within the continuation object. The continuation object stores an instruction pointer, which can be freely copied. However, we have to update the saved stack pointer to point to the duplicate copy of the saved stack. This is sufficient for Wasmtime, but other implementations may require extra work. For example, if an implementation stores pointers into the stack in registers or on the stack itself, they must be updated to point to the copy.

6 Evaluation

In this section we compare `Wasm/k` to the natural alternative: which is to implement continuations using a whole program transformation that doesn't require any change to WebAssembly.

Asyncify [28] is a tool that simulates non-blocking I/O in WebAssembly. It extends WebAssembly with control operators that are similar to one-shot continuations, and outputs

[6] Our implementation is available at https://wasmk.github.io.
[7] Commit 21124ee of the C-Ray fork available at https://wasmk.github.io was used in this experiment.

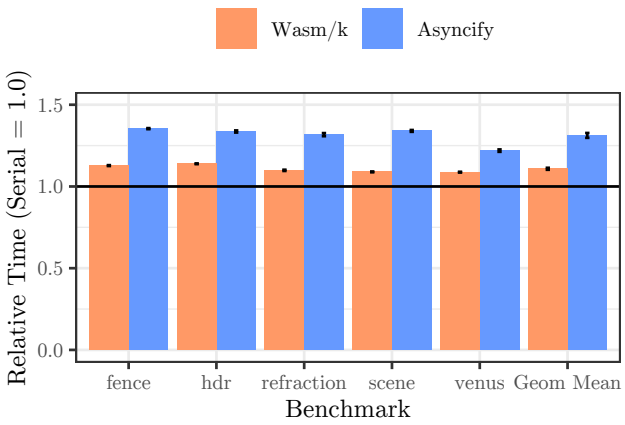


Figure 12. Performance of green threads implemented using `Wasm/k` and Asyncify in a ray tracing application [7]. All experiments were performed on a 64 bit 3.3GHz 4-Core CPU on Ubuntu. Error bars show the 95% confidence interval of the overhead, over six trials.

standard WebAssembly that simulates control flow. We use both `Wasm/k` and Asyncify to implement a green threading library (Section 2.4), which allows us to directly compare the performance of threaded programs.

As a benchmark, we use C-Ray [1], which is a ray tracer implemented in approximately 9,500 lines of C. Ray tracers are compute-intensive, and take a long time to render the final, full-quality image. However, because they can compute lower quality rendering approximations incrementally, it should be possible to display incremental rendering results, to appear more responsive to the user. C-Ray performs rendering computations on background threads (using `pthreads`), while the main thread periodically displays the current scene. We ported C-Ray to use our green threading library, and inserted thread yields in the main rendering loop, which yielded approximately once every 25 ms.

Code size. Code size is particularly important when running WebAssembly in web browsers, which download code on demand, and a key factor of WebAssembly's design is that it has a compact binary file format. The size of the C-Ray WebAssembly program is 1.3× larger with Asyncify than it is with `Wasm/k`.

Performance. Figure 12 shows the time needed to complete ray tracing on five different visual scenes using `Wasm/k` and Asyncify, with the geometric mean over all scenes shown in the last two columns on the right. As a baseline, we use C-Ray running in WebAssembly with no threading. Note that the baseline has limited utility, since it cannot show intermediate results. However, it does illustrate the overhead that both `Wasm/k` and Asyncify introduce. The mean running time of `Wasm/k` is 1.1× the running time without threads. In contrast, the mean running time of Asyncify is 1.3× the running time without threads. Asyncify is slower, since it introduces several loads, stores, and branches to the compiled code. The smaller slowdown that `Wasm/k` introduces is the cost of checking whether it is time to switch threads.

7 Related Work

WebAssembly. `Wasm/k` extends the formal semantics of WebAssembly 1.0 [18]. There are several proposed extensions to WebAssembly 1.0, not all of which have been implemented in production web browsers. The threading proposal [8] extends WebAssembly with support for atomic memory operations and synchronization primitives, but leaves the API for thread creation up to each WebAssembly runtime implementation. Thus far, the `pthread` API has been supported in some browsers. Watt et al. build on the threading proposal by formalizing a semantics and memory model for concurrent WebAssembly [27]. This work on robust support for concurrency via physical threads is an important step for WebAssembly, and is orthogonal and complimentary to `Wasm/k`: both aspects are needed for efficient implementations of goroutines which can utilize all CPU cores.

Another proposal extends WebAssembly with support for exception handling [3], which is a form of limited stack manipulation. An interesting question of semantics not addressed in this work is how `Wasm/k` would interact with exception handling. In this direction, there is prior work on supporting both delimited continuations and exception handling [16].

An alternative to supporting continuations natively is to implement them by source-to-source transformation [12, 22]. Asyncify [28] does so for WebAssembly, and the Go compiler uses a similar approach to support Goroutines. Our evaluation (Section 6) shows that `Wasm/k` is significantly faster than source-to-source transformation, and produces smaller programs.

A recent discussion sketched an alternative design for WebAssembly continuations [24] that is based on extending exception handlers with general effect handlers. Our design is orthogonal to exception handling and makes fewer changes to the WebAssembly 1.0 type system. To the best of our knowledge, this alternative design has not been implemented at this time.

Finally, there exist related strategies of program execution control. Existing interpreters or virtual machines which feature execution control mechanisms can be compiled to WebAssembly, such as the Lua VM (implemented in C) [10] which features coroutines. This is certainly a viable and straightforward strategy to allow stack-manipulating code to run in a WebAssembly environment, but may not be able to achieve performance comparable to compiling to WebAssembly. In addition, debuggers can be seen as a form of execution control, as code can be paused and resumed, but

unlike with first-class continuations, the program control is not internally observable. Debugger support for WebAssembly has recently been explored in the context of microcontrollers [17].

Continuations. We adapt Sitaram and Felleisen's *control* operator [25] for WebAssembly. Our design accounts for the fact that WebAssembly has neither first-class functions, nor garbage collection: programs must explicitly delete unused continuations, and our new control operators take additional arguments that are not necessary in languages that support closures. We rely on control delimiters to ensure that WebAssembly programs always safely interoperate with host languages that do not support continuations, such as JavaScript. However, it should be possible to adapt other control operators as well [13, 15].

A goal of Wasm/k is to show that delimited continuations can be implemented efficiently in a modern WebAssembly JIT. Our implementation uses a contiguous stack, since it does not require global changes to code generation. However, there are a variety of other implementation strategies with different tradeoffs [14].

8 Conclusion

We have presented Wasm/k, an extension to WebAssembly that adds support for delimited one-shot continuations with explicit copying. We have prototyped all phases of Wasm/k, with examples in C/C++, code generation from C/C++ to Wasm/k, formal semantics of Wasm/k, and an efficient implementation of Wasm/k in an existing JIT. We hope that Wasm/k is a step toward helping WebAssembly be an effective compilation target for a large variety of high-level languages.

Acknowledgements

This work was partially supported by the National Science Foundation under grants CCF-2007066, CCF-1453474, and CCF-1564162.

References

[1] 2020. C-Ray. https://github.com/vkoskiv/c-ray. Accessed Jul 4 2020.
[2] 2020. Electron. https://www.electronjs.org. Accessed July 5, 2020.
[3] 2020. Exception handling. https://github.com/WebAssembly/exception-handling/blob/master/proposals/Exceptions.md. Accessed Jul 4 2020.
[4] 2020. generator.rkt. https://github.com/racket/racket/blob/ac4ae9ebba653c76edb2bb1f08ec1007427e5333/racket/collects/racket/generator.rkt#L30. Accessed July 5, 2020.
[5] 2020. misc/Wasm: long tasks with Go WebAssembly. https://github.com/golang/go/issues/39620. Accessed July 5, 2020.
[6] 2020. runtime: fatal error: self deadlock WebAssembly. https://github.com/golang/go/issues/35256. Accessed July 5, 2020.
[7] 2020. syscall/js: performance considerations. https://github.com/golang/go/issues/32591. Accessed July 5, 2020.
[8] 2020. Threading proposal for WebAssembly. https://github.com/WebAssembly/threads/blob/master/proposals/threads/Overview.md. Accessed Jul 4 2020.
[9] 2020. Wasm: 3x performance overhead of using WebAssembly in Node 8. https://github.com/golang/go/issues/26277. Accessed July 5, 2020.
[10] 2020. wasm_lua. https://github.com/vvanders/wasm_lua. Accessed Sept 6, 2020.
[11] 2020. WebAssembly Specification. https://webassembly.github.io/spec/core/. Accessed Jul 6 2020.
[12] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. 2018. Putting in All the Stops: Execution Control for JavaScript *(PLDI'18)*.
[13] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control *(LFP'90)*. 151–160.
[14] Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations *(PLDI'20)*.
[15] Mattias Felleisen. 1988. The Theory and Practice of First-Class Prompts *(POPL'88)*. 180–190.
[16] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment *(ICFP'07)*.
[17] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers *(MPLR'2019)*.
[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly *(PLDI'17)*.
[19] Xuan Huang. 2020. A Mechanized Formalization of the WebAssembly Specification in Coq. https://www.cs.rit.edu/~mtf/student-resources/20191_huang_mscourse.pdf. Accessed Jul 7 2020.
[20] Abhinav Jangda, Bobby Powers, Emery Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code *(ATC'19)*.
[21] Brooks Paige and Frank Wood. 2014. A Compilation Target for Probabilistic Programming Languages *(ICML'14)*.
[22] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. 2005. Continuations from generalized stack inspection *(ICFP'05)*.
[23] Donald Pinckney, Arjun Guha, and Yuriy Brun. 2020. Wasm/k: Delimited Continuations for WebAssembly. https://arxiv.org/abs/2010.01723
[24] Andreas Rossberg, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, Sam Lindley, and Stephen Dolan. 2019. Stacks and Continuations for Wasm — Idea Sketch. https://github.com/WebAssembly/meetings/blob/master/main/2020/presentations/2020-02-rossberg-continuations.pdf. Accessed Jul 4 2020.
[25] Dorai Sitaram and Matthias Felleisen. 1990. Control Delimiters and Their Hierarchies. *LISP and Symbolic Computation* 3, 1 (May 1990), 67–99.
[26] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification *(CPP'18)*.
[27] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly *(OOPSLA'19)*.
[28] Alon Zakai. 2019. Pause and Resume WebAssembly with Binaryen's Asyncify. https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html. Accessed Jul 4 2020.