Wasm/k: Delimited Continuations for WebAssembly

Donald Pinckney
Northeastern University
USA

Arjun Guha
Northeastern University
USA

Yuriy Brun
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 Wasm-time, 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

ACM Reference Format:

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 1.5× 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 (Coroutines), 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 exceptions1; 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 [27] 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 [27]. 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.

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 [27].

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 Wasm-time, 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 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. 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(20), 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 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

---

This design ensures that a malicious program cannot alter return addresses to escape the WebAssembly sandbox.
shows an alternate implementation of line shows an implementation of line is a small program that uses this threading library.

Wasm/k 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.

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 an implementation of green threads in C/k, 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 dequeuing 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 does not support first-class functions. Therefore, we re-present 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 need for 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.

**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

```cpp
// Allocating a generator
Generator *make_generator(void (*f)(Generator *)) { GeneratedGenerator *g = (GeneratedGenerator *)malloc(sizeof(GeneratedGenerator)); g->after_yield = convertFuncToCont(f); return g;
}

// Yielding implementation
void yield_handler(k_id k, Generator *g) { g->after_yield = k; restore(g->after_yield, g->value); }

// Next implementation
void next_handler(k_id k, Generator *g) { g->after_next = k; restore(g->after_next, g->value); }

// Freeing a generator
void free_generator(Generator *g) { continuation_delete(g->after_yield); free(g); }
```

**Figure 4.** Green threads in C/k

**Figure 5.** Generators in C/k.
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 \( \text{uniform}(1, 2, 3) \) 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 `ContinuationThunk` (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 15) 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.

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
The nested structure of labelled blocks is shown in Figure 7a and the execution trace of our example.

We present a small-step semantics that updates the stack, and possibly the store, with a boundary that only encodes loops, and can be ignored in this paper.

The global execution state of all instances is called the store. Moreover, the semantics represents the 64-bit value $n$ as $(\text{\texttt{i64.const}} n)$. 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., \texttt{block . . . end} and \texttt{loop . . . end}). The WebAssembly semantics turns all kinds of blocks into labelled blocks (label{e*} . . . end), which are an administrative instruction.\footnote{The e* is only needed to encode loops, and can be ignored in this paper.} 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.

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_i) at each step (s; v_i; e* → s’; v_i’; e’). The semantics is congruent with local contexts: if s; v_i; e* → s’; v_i’; e’ then s; v_i; L^k[e*] → s’; v_i’; 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.

![Figure 7](image-url)  
**Figure 7.** An example of WebAssembly execution.

![Figure 8](image-url)  
**Figure 8.** Syntax of Wasm/k: we extend the WebAssembly runtime structure with a table of continuations.

### 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 \texttt{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 \((\kappa_\text{id})\). 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 4). Each instance \(\text{inst}\) contains a stack of dynamically nested prompt contexts. Each prompt context \(\text{pinst}\) 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 \(\text{nil}\) or a captured continuation, with all entries initialized to \(\text{nil}\).

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 \(L^{\text{max}}\) 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, \(\text{root}\) will be set to \(\text{nil}\), otherwise \(\text{root}\) will be set to the index \(\kappa_R\) of the root continuation in the continuation table.

New reduction rules. The semantics of WebAssembly define a reduction relation \((\rightarrow)\) 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 \((\rightarrow^*)\) for programs that contain \(\text{control } h\) and \(\text{restore}\) instructions (Figure 9). The extended semantics refer to the original WebAssembly reduction relation \((\rightarrow)\), using the [Cong] rule, but there is no equivalent rule for \((\rightarrow^*)\). If there is a reduction which involves no use of \(\text{control } h\) or \(\text{restore}\), then it is also a valid reduction which might make use of \(\text{control } h\) or \(\text{restore}\), 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 \(\kappa\) and the argument \(v\). The continuation ID is bound to the current continuation \((L^{\text{max}})\) and local variables \(v^i\), 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 \(\kappa\) and a restore value \(v\). The instruction dynamically checks that \(\kappa\) is a valid continuation ID. If \(\kappa\) is valid, it restores the local variables \(v^i\) and the stack \((L^{\text{max}})\) that is associated with \(\kappa\), and returns \(v\) to the stack. The restore instruction also marks the continuation ID \(\kappa\) as \(\text{nil}\) 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 \(\text{nil}\), 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 \(\kappa\) 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 \(\text{nil}\). 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 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 \(\text{nil}\), 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

---

3To 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.

5A type annotation of the body \(e^*\) of the prompt, and is not important to understand the semantics.
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 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` (if `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

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.
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 \([18, 19, 25]\), which we build on.

WebAssembly’s instruction typing relation \((\Rightarrow e^*; t^*)\) 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 \((\Rightarrow)\). However, if a step captures or restores a continuation \((\leadsto)\), the type of the current instruction sequence may change.

To address this, we introduce a new typing relation \((\Rightarrow^k)\) 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 \(([\text{Root}])\), and 2) if not, we extract the type of the saved primary root stack from the store \(([\text{Non-Root}])\). Using this typing relation, we prove progress and preservation for Wasm/k.

**Theorem 3.1** \((\leadsto)\) Preservation. If \(\Rightarrow^k s; v^*; e^*: t^* \text{ and } s; v^*; e^*: \delta^* \leadsto s' \delta'; v'^*; e'^* \Rightarrow \delta^* \leftrightarrow s; v^*; e^*: t^*\).

**Theorem 3.2** \((\Rightarrow)\) Progress. If \(\Rightarrow^k s; v^*; e^*: t^* \text{, then either } e^* = v'^* \text{ or } e^* = \text{trap or } s; v^*; e^*: \delta^* \leadsto s'; v'^*; e'^* : t^*\).

The proofs of both theorems are available in the appendix.

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.

Figure 12. Performance of green threads implemented using Wasm/k and Asyncify in a ray tracing application. 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.

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.

---

6Our implementation is available at https://wasmk.github.io.
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 [27] 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 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 for 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 $1.3 \times$ 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 $1.1 \times$ the running time without threads. In contrast, the mean running time of Asyncify is $1.3 \times$ 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 and 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 [26]. 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 [27] does so for WebAssembly, and the Go compiler uses a similar approach to supports 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 [23] 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 [24] 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

---

Commit 21124ee of the C-Ray fork available at https://wasmk.github.io was used in this experiment.
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
A Appendix

A.1 Continuation Table Typing Relation

Figure 11 presented the core rules for the continuation table typing relation \( i^k \). Figure 11 relies on an extension of the WebAssembly store typing, in which a store \( s \) is given a type \( S \), now extended with the pstack field as shown in Figure 10. Modifications to existing WebAssembly store typing rules and additional rules are required to compute the pstack field of \( S \). These modifications and new rules are given in Figure 13, where modified WebAssembly rules are marked with a \( (†) \).

\[
\begin{array}{l}
(S \triangleright cl : tf) \triangleright (\triangleright v : t) \triangleright (S_{ab}(t) = n) \triangleright (S_{mem}(J) = m) \triangleright (S \triangleright e \triangleright v^*; L_{max}[\{i64, \text{const } 0\}]; tk^k) \triangleright (\text{itable}(k_R) \neq \text{nild}) \triangleright \\
S \triangleright \{\text{func } t^e, \text{glob } v^*, \text{tab } t^e, \text{mem } J, \text{pstack} \{\text{itable}([\{\text{locals } v^*, \text{ctx } L_{max}, \text{inst } c\} ; \text{nild}], \text{root}(k_R \mid \text{nild})^*) \}
\end{array}
\]

(Prompt-End)

\[
S_{\text{inst}}(i)_{\text{pstack}(0)_{\text{root}} = \text{nild}} \\
S; C \triangleright i \triangleright \text{prompt end} : e \rightarrow e
\]

Figure 13. Additional Store Typing Rules.

In addition, we define the shorthand notation \( S_{\text{inst}}(i)_{\text{roots}} \) used in Figure 11 to be the vector \( S_{\text{inst}}(i)_{\text{pstack}(\cdot)_{\text{root}}} \).

A.2 Proofs of Type Safety Properties

Lemma A.1 (Context Substitution). If

1. \( \triangleright i; s; v^*; L_i^k[e^*] : t^* \), and
2. \( \triangleright i; s; v^*; e^* : t^* \), and
3. \( \triangleright i; s; v^*; e^* : \tilde{t}^e \),

then \( \triangleright i; s; v^*; L_i^k[e^*] : t^* \).

Proof. This is a direct consequence of the transitive rule for WebAssembly type checking. \( \square \)

Theorem A.2 (\( \rightsquigarrow \) Preservation). If \( \triangleright i^k \triangleright s; v^*; e^* : t^* \) and \( s; v^*; e^* \rightsquigarrow s'; v'^*; e'^* \), then \( \triangleright i^k \triangleright s'; v'^*; e'^* : t^* \).

Proof. Suppose that:

H1) \( \triangleright i^k \triangleright s; v^*; e^* : t^* \)
H2) \( s; v^*; e^* \rightsquigarrow s'; v'^*; e'^* \)

We want to show that \( \triangleright i^k \triangleright s'; v'^*; e'^* : t^* \).

By H1 and the typing rules [Root] and [Non-Root] for \( i^k \), we know:

H3) \( \triangleright s : S \)

Based on the [Root] and [Non-Root] typing rules for H1, there are two cases:

Case C1 \( S_{\text{inst}}(i)_{\text{roots}} = \text{nild} \). In this case we also know:

H4) \( \triangleright i; s; v^*; e^* : t^* \)

By H2 there are 10 subcases to consider:

Case C1.1 \( s; v^*; L_{\text{max}}[\{\text{i64, const } v\}](\text{control } h) \rightsquigarrow s'; \epsilon; (\text{i64, const } k)(\text{i64, const } v)(\text{call } h) \text{ trap } \wedge (s', k) = \delta_{ctu}(s, i, v^*, L_{\text{max}}) \)

By Case C1 and C1.1:

H5) \( \delta_{ctu}(s, i, v^*, L_{\text{max}}) = (s', k) = (\text{setCont(setRoot(s, i, k)), i, k, }{\text{locals } = v^*, \text{ctx } L_{\text{max}}}, k) \)
H6) \( \text{getCont(s, i, k) = nil} \)

By H3 and H5 we can type check \( s' \):

H7) \( \triangleright s' : S' \)
H8) \( S'_{\text{inst}}(i)_{\text{pstack}(0)_{\text{root}} = k} \)
H9) \( \forall p \geq 1, S'_{\text{inst}}(i)_{\text{pstack}(p)_{\text{root}}} = \text{nild} \)

We can compute the type of the stored stack in \( s' \). By Lemma A.1, H4 and H5:

H10) \( S'_{\text{inst}}(i)_{\text{pstack}(0)_{\text{itable}}(k) = \text{t}^*} \)

The new stack can be independently type checked. By the control type checking rule, \( S'_{\text{inst}}(i)_{\text{func}(h) = \text{i64 i64} \rightarrow \epsilon} \). Thus,

H11) \( \triangleright s'; \epsilon; (\text{i64, const } k)(\text{i64, const } v)\text{(call } h)\text{ trap} : \epsilon \)

By H7, H8, H9, H10, H11 and the [Non-Root] rule:
H12) \( i^+ s'; \epsilon; (i64\text{.const } k)(i64\text{.const } v)(\text{call } h)\text{trap} : t^* \)

Case C1.1 is complete.

Case C1.2) \( s; v^*; L^{\max}[(i64\text{.const } k)(i64\text{.const } v)\text{restore}] \sim \rightarrow s'; v'^*; L^{\max'}[(i64\text{.const } v)] \wedge (s', v^*, L^{\max'}) = \delta_{\text{rest}}(s, i, k) \)

By C1.2 getRoot(s, i) \neq \text{nil}. However this is a contradiction with case C1 which implies getRoot(s, i) = \text{nil}. Therefore, Case C1.2 is impossible.

Case C1.3) \( s; v^*; L^{\max}[(i64\text{.const } k)(i64\text{.const } v)\text{restore}] \sim \rightarrow s; v'^*; \text{trap} \wedge \delta_{\text{rest}}(s, i, k) \) \text{undefined}.

By the type checking of \text{trap},

H13) \( i s; v'; \text{trap} : t^* \)

By Case C1, H3, H13, and the [Root] rule, we conclude:

H14) \( i^+ s; v'; \text{trap} : t^* \)

Case C1.3 is complete.

Case C1.4) \( s; v^*; L^k[(i64\text{.const } k')\text{continuation_copy}] \leftarrow i s; v'^*; L^k[(i64\text{.const } k')] \wedge (s', k') = \delta_{\text{copy}}(s, i, k) \)

By Case C1.4:

H15) \( \delta_{\text{copy}}(s, i, k) = (s', k') = (\text{setCont}(s, i, k', \text{getCont}(s, i, k)), k') \)

H16) getCont(s, i, k') = \text{nil}.

By H4, Case C1.4, and Lemma A.1,

H17) \( i s'; v'; L^k[(i64\text{.const } k')] : t^* \)

By H3, there exists a type for the stack that is being copied:

H18) \( \exists r'^\ast, S_{\text{inst}}(i)_{\text{pstack}}(0)_{\text{table}}(k) = t'^\ast \)

By Case C1, H15, H18, and the store typing relation,

H19) \( i s': S' \)

H20) \( S'_{\text{inst}}(i)_{\text{roots}} = \text{nil}' \)

By H17, H19, H20 and the [Root] rule:

H21) \( i^+ s'; v'^*; L^k[(i64\text{.const } k')] : t^* \)

Case C1.4 is complete.

Case C1.5) \( s; v^*; L^k[(i64\text{.const } k)\text{continuation_copy}] \leftarrow i s; v'^*; L^k[\text{trap}] \wedge \delta_{\text{copy}}(s, i, k) \) \text{undefined}

By the type checking of \text{trap}, H4, and Lemma A.1,

H22) \( i s; v'; L^k[\text{trap}] : t^* \)

By Case C1, H3, H22, and the [Root] rule, we conclude:

H23) \( i^+ s; v'; L^k[\text{trap}] : t^* \)

Case C1.5 is complete.

Case C1.6) \( s; v^*; L^k[(i64\text{.const } k)\text{continuation_delete}] \leftarrow i s'; v'^*; L^k[e] \wedge s' = \delta_{\text{delete}}(s, i, k) \)

By Case C1.6:

H24) \( \delta_{\text{delete}}(s, i, k) = s' = \text{setCont}(s, i, k, \text{nil}) \)

By H4, Case C1.6, and Lemma A.1,

H25) \( i s'; v'; L^k[e] : t^* \)

By Case C1, H24, and the store typing relation,

H26) \( i s': S' \)

H27) \( S'_{\text{inst}}(i)_{\text{roots}} = \text{nil}' \)

By H25, H26, H27 and the [Root] rule:

H28) \( i^+ s'; v'^*; L^k[e] : t^* \)

Case C1.6 is complete.

Case C1.7) \( s; v^*; L^k[(i64\text{.const } k)\text{continuation_delete}] \leftarrow i s; v'^*; L^k[\text{trap}] \wedge \delta_{\text{delete}}(s, i, k) \) \text{undefined}

By the type checking of \text{trap}, H4, and Lemma A.1,

H29) \( i s; v'; L^k[\text{trap}] : t^* \)

By Case C1, H3, H29, and the [Root] rule, we conclude:

H30) \( i^+ s; v'^*; L^k[\text{trap}] : t^* \)

Case C1.7 is complete.

Case C1.8) \( s; v^*; L^k[\text{prompt tf e' end}] \leftarrow i s'; v'^*; L^k[\text{block tf e' end prompt_end}] \wedge s' = \delta_{p}(s, i) \)

By Case C1.8:

H31) \( \delta_{p}(s, i) = s' \) where \( s' = s \) except \( s'_{\text{inst}}(i)_{\text{pstack}} \leftrightarrow \text{push}(s_{\text{inst}}(i)_{\text{pstack}}, \{\text{etable} = \text{nil}'\}, \text{root} = \text{nil}') \)

By H4, H31, the type checking of \text{prompt}, and Lemma A.1,

H32) \( i s'; v'^*; L^k[\text{block tf e' end prompt_end}] : t^* \)
By H3, H31 and store typing:
\[ H33 \vdash s' : S' \]
\[ H34 \quad S_{\text{inst}}(\text{roots}) = \text{nil} \]
By H32, H33, H34 and the [Root] rule:
\[ H35 \quad \vdash s'; \nu^*; L^k[\text{block } if \ e^* \text{ end prompt_end}] : t^* \]

Case C1.8 is complete.

**Case C1.9**
\[ s'; \nu^*; L^k[\text{prompt_end}] \rightarrow t \quad s'; \nu^*; L^k[e] \land s' = \delta_{\text{p-end}}(s, i) \]

By Case C1.9:
\[ H36 \quad \delta_{\text{p-end}}(s, i) = s' \quad \text{ where } s' = s \quad \text{ except } s'_{\text{inst}}(i)_{\text{pstack}} \rightarrow \text{pop}(s_{\text{inst}}(i)_{\text{pstack}}) \]

By H4, H36, the type checking of \text{prompt_end}, and Lemma A.1,
\[ H37 \quad \vdash s'; \nu^*; L^k[e] : t^* \]

By H3, H36 and store typing:
\[ H38 \quad \vdash s' : S' \]
\[ H39 \quad S_{\text{inst}}(\text{roots}) = \text{nil} \]
By H37, H38, H39 and the [Root] rule:
\[ H40 \quad \vdash s'; \nu^*; L^k[e] : t^* \]

Case C1.9 is complete.

**Case C1.10**
\[ s; \nu^*; e^* \rightarrow t \quad s'; \nu''^*; e''^* \quad \text{ for some redex in standard WebAssembly.} \]
Since standard WebAssembly redexes do not modify the continuation tables, \( \vdash s' : S' \) where \( S'_{\text{inst}}(\text{roots}) = \text{nil} \).
Therefore, by H4, the Preservation theorem of standard WebAssembly, and the [Root] rule:
\[ H41 \quad \vdash s'; \nu''^*; e''^* : t^* \]

Case C1.10 is complete.

**Case C2**
\[ \exists r, s.t. \quad p_r = \max\{p \mid S_{\text{inst}}(p_{\text{inst}}(p)_{\text{root}} \neq \text{nil}) \} \quad \text{in this case we also know:} \]
\[ H42 \quad o_R = S_{\text{inst}}(p_{\text{inst}}(p)_{\text{root}}) \]
\[ H43 \quad \exists r, \nu^*; s; \nu^*; e^* : t^* \]
\[ H44 \quad t^* = S_{\text{inst}}(p_{\text{inst}}(p)_{\text{table}}(\kappa_R)) \]

By H2 there are 10 subcases to consider:

**Case C2.1**
\[ s; \nu^*; L_{\text{max}}[i64.\text{const}(v)\text{(control } h)] \rightarrow s; \nu^*; e; (i64.\text{const } k) \quad (i64.\text{const } v) \quad (\text{call } h) \quad \text{trap} \land (s', \nu^*; L_{\text{max}}) \]
There are two sub-sub cases, either getRoot(s, i) = nil or getRoot(s, i) = \( k_0 \) for some \( k_0 \):

**Case C2.1.1**
\[ \text{getRoot}(s, i) = \text{nil} \land p_r \neq 0 \]

By Case C2.1 and C2.1.1:
\[ H45 \quad \delta_{\text{ct}}(s, i, \nu^*, L_{\text{max}}) = (s', \kappa) = \text{(setCont}(	ext{getRoot}(s, i, k), i, k, \{\text{locals} = \nu^*_i, \text{ctx} = L_{\text{max}}\}), k) \]
\[ H46 \quad \text{getCont}(s, i, k) = \text{nil} \]

By H3 and H45 we can type check \( s' \):
\[ H47 \quad \vdash s' : S' \]
\[ H48 \quad p_r = \max\{p \mid S'_{\text{inst}}(p_{\text{inst}}(p)_{\text{root}} \neq \text{nil}) \} \]
\[ H49 \quad S'_{\text{inst}}(p_{\text{inst}}(p)_{\text{root}}) = \kappa_R \]
\[ H50 \quad S'_{\text{inst}}(p_{\text{inst}}(p)_{\text{table}}(k_R)) = t^* \]
The new stack can be independently type checked. By the control type checking rule, \( S'_{\text{inst}}(p_{\text{inst}}(h) = i64 \rightarrow e \).

Thus,
\[ H51 \quad \vdash s'; e; (i64.\text{const } k)(i64.\text{const } v)(\text{call } h)\text{trap} : e \]
By H47, H48, H49, H50, H51 and the [Non-Root] rule:
\[ H52 \quad \vdash s'; e; (i64.\text{const } k)(i64.\text{const } v)(\text{call } h)\text{trap} : t^* \]
Case C2.1.1 is complete.

**Case C2.1.2**
\[ \text{getRoot}(s, i) = k_0 \neq \text{nil} \]

By Case C2.1 and C2.1.2:
\[ H53 \quad \delta_{\text{ct}}(s, i, \nu^*, L_{\text{max}}) = (s', \kappa) = \text{(setCont}((s, i, k, \{\text{locals} = \nu^*_i, \text{ctx} = L_{\text{max}}\}), k) \]
\[ H54 \quad \text{getCont}(s, i, k) = \text{nil} \]

By H3 and H53 we can type check \( s' \):
\[ H55 \quad \vdash s' : S' \]
\[ H56 \quad p_r = \max\{p \mid S'_{\text{inst}}(p_{\text{inst}}(p)_{\text{root}} \neq \text{nil}) \} \]
\[ H57 \quad S'_{\text{inst}}(p_{\text{inst}}(p)_{\text{root}}) = \kappa_R \]
\[ H58 \quad S'_{\text{inst}}(p_{\text{inst}}(p)_{\text{table}}(k_R)) = t^* \]
The new stack can be independently type checked. By the control type checking rule, 
\[ S'_{\text{inst}}(i)_{\text{func}}(h) = \text{i64} \rightarrow \epsilon. \]
Thus,
\[
H59) \vdash^r s'; \epsilon; (\text{i64}.\text{const} \ k)(\text{i64}.\text{const} \ v)(\text{call h})\text{trap} : \epsilon
\]
By \( H55, H56, H57, H58, H59 \) and the [Non-Root] rule:
\[
H60) \vdash^r s'; \epsilon; (\text{i64}.\text{const} \ k)(\text{i64}.\text{const} \ v)(\text{call h})\text{trap} : t^*
\]
Case C2.1.2 is complete.

**Case C2.2** \( s; v'; L^\max[\text{i64}.\text{const} \ k)(\text{i64}.\text{const} \ v)\text{restore}] \rightsquigarrow s'; v''; L^\max[\text{i64}.\text{const} \ v] \wedge (s', v'', L^\max) = \delta_{\text{rest}}(s, i, k) \)

By C2.2:

\( H61 \) getRoot(s, i) = \( \kappa_0 \neq nil \)

There are two sub-sub cases, either \( \kappa = \kappa_0 \) or \( \kappa \neq \kappa_0 \):

**Case C2.2.1** \( \kappa = \kappa_0 \)

By C2.2.1:

\( H62) \delta_{\text{rest}}(s, i, \kappa) = (s', v'', L^\max) = (\text{setRoot(setCont(s, i, \kappa, nil), i, nil), getCont(s, i, \kappa)_{\text{locals}}, getCont(s, i, \kappa)_{\text{ctx}}}) \)

By \( H3 \) and \( H62 \) we can type check \( s' \):

\( H63) \vdash s' : S' \)

There are two sub-sub sub cases, either \( p_r = 0 \) or \( p_r \geq 1 \):

**Case C2.2.1.1** \( p_r = 0 \)

By C2.2.1.1 and \( H42 \):

\( H64) \kappa = \kappa_0 = \kappa_R = \text{S}_{\text{inst}}(i)_{\text{pstack}}(p_R)_{\text{root}} \)

By C2.2.1.1, \( H62 \), and \( H63 \):

\( H65) \text{S}_{\text{inst}}(i)_{\text{roots}} = \text{nil}^* \)

By C2.2.1.1, \( H64 \), and Lemma A.1:

\( H66) \vdash^r s'; v''; L^\max[\text{i64}.\text{const} \ v] : t^* \)

By \( H63, H65, H66 \), and the [Root] rule:

\( H67) \vdash^r s'; v''; L^\max[\text{i64}.\text{const} \ v] : t^* \)

Case C2.2.1.1 is complete.

**Case C2.2.1.2** \( p_r \geq 1 \)

By C2.2.1.2 and \( H63 \):

\( H68) p_r = \max\{p \mid \text{S}_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} \neq \text{nil}\} \)

\( H69) \text{S}_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} = \kappa_R = \kappa_r \)

\( H70) \text{S}_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{table}}(\kappa_R) = t^* \)

The stack we are switching to can be type checked by \( H3 \) and Lemma A.1:

\( H71) \exists \delta''', \vdash^r s'; v''; L^\max[\text{i64}.\text{const} \ v] : t^* \)

By \( H63, H68, H69, H70, H71 \), and the [Non-Root] rule:

\( H72) \vdash^r s'; v''; L^\max[\text{i64}.\text{const} \ v] : t^* \)

Case C2.2.1.2 is complete.

**Case C2.2.2** \( \kappa \neq \kappa_0 \)

By C2.2.2:

\( H73) \delta_{\text{rest}}(s, i, \kappa) = (s', v'', L^\max) = (\text{setCont(s, i, k, nil), getCont(s, i, k)_{\text{locals}}, getCont(s, i, k)_{\text{ctx}}}) \)

By \( H3 \) and \( H73 \) we can type check \( s' \):

\( H74) \vdash s' : S' \)

\( H75) p_r = \max\{p \mid \text{S}_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} \neq \text{nil}\} \)

\( H76) \text{S}_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} = \kappa_R \)

\( H77) \text{S}_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{table}}(\kappa_R) = t^* \)

The stack we are switching to can be type checked by \( H3 \) and Lemma A.1:

\( H78) \exists \delta''', \vdash^r s'; v''; L^\max[\text{i64}.\text{const} \ v] : t^* \)

By \( H74, H75, H76, H77, H78 \), and the [Non-Root] rule:

\( H79) \vdash^r s'; v''; L^\max[\text{i64}.\text{const} \ v] : t^* \)

**Case C2.3** \( s; v'; L^\max[\text{i64}.\text{const} \ k)(\text{i64}.\text{const} \ v)\text{restore}] \rightsquigarrow s; v'; \text{trap} \wedge \delta_{\text{rest}}(s, i, k) \) undefined

By the type checking of \( \text{trap} \),

\( H80) \vdash s; v'; \text{trap} : \epsilon \)

By Case C2, \( H3, H42, H44, H80 \), and the [Non-Root] rule, we conclude:

\( H81) \vdash s; v'; \text{trap} : t^* \)
Case C2.3 is complete.

Case C2.4 $s; v_1; L^k[(i64.const \ k) \text{continuation\_copy}] \rightarrow_i s'; v'; L^k[(i64.const \ k')] \land (s', k') = \delta_{\text{copy}}(s, i, k)$

By Case C2.4:

H82 $\delta_{\text{copy}}(s, i, k) = (s', k') = (\text{setCont}(s, i, k', \text{getCont}(s, i, k)), k')$

H83 $\text{getCont}(s, i, k) = \text{nil}$.

By H43, Case C2.4, and Lemma A.1,

H84 $\vdash_i s'; v'; L^k[(i64.const \ k')] : \mathcal{I}^*$

By H3, there exists a type for the stack that is being copied:

H85 $\exists t^*, S_{\text{inst}}(i)_{\text{pstack}}(0)_{\text{table}}(k) = t^*$

By Case C2, H82, H83, H85, and the store typing relation:

H86 $\vdash s': S'$

H87 $p_r = \max\{p \mid S'_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} \neq \text{nil}\}$

H88 $S'_{\text{inst}}(i)_{\text{pstack}}(p_r)_{\text{root}} = \kappa_R$

H89 $S'_{\text{inst}}(i)_{\text{pstack}}(p_r)_{\text{table}}(\kappa_R) = t^*$

By H84, H86, H87, H88, H89 and the [Non-Root] rule:

H90 $\vdash_i s'; v'; L^k[(i64.const \ k')] : \mathcal{I}^*$

Case C2.4 is complete.

Case C2.5 $s; v_1; L^k[(i64.const \ k) \text{continuation\_copy}] \rightarrow_i s; v; L^k[\text{trap}] \land \delta_{\text{copy}}(s, i, k) \text{ undefined}$

By the type checking of trap, H43, and Lemma A.1,

H91 $\vdash_i s; v; L^k[\text{trap}] : \mathcal{I}^*$

By Case C2, H3, H42, H44, H91, and the [Non-Root] rule, we conclude:

H92 $\vdash_i s; v; L^k[\text{trap}] : \mathcal{I}^*$

Case C2.5 is complete.

Case C2.6 $s; v_1; L^k[(i64.const \ k) \text{continuation\_delete}] \rightarrow_i s'; v'; L^k[\epsilon] \land s' = \delta_{\text{delete}}(s, i, k)$

By Case C2.6:

H93 $\delta_{\text{delete}}(s, i, k) = s' = \text{setCont}(s, i, k, \text{nil})$

By H43, Case C2.6, and Lemma A.1,

H94 $\vdash_i s'; v'; L^k[\epsilon] : \mathcal{I}^*$

By Case C2, H93, and the store typing relation:

H95 $\vdash s'; S'$

H96 $p_r = \max\{p \mid S'_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} \neq \text{nil}\}$

H97 $S'_{\text{inst}}(i)_{\text{pstack}}(p_r)_{\text{root}} = \kappa_R$

H98 $S'_{\text{inst}}(i)_{\text{pstack}}(p_r)_{\text{table}}(\kappa_R) = t^*$

By H94, H95, H96, H97, H98 and the [Non-Root] rule:

H99 $\vdash_i s'; v'; L^k[(i64.const \ k')] : \mathcal{I}^*$

Case C2.6 is complete.

Case C2.7 $s; v_1; L^k[(i64.const \ k) \text{continuation\_delete}] \rightarrow_i s; v; L^k[\text{trap}] \land \delta_{\text{delete}}(s, i, k) \text{ undefined}$

By the type checking of trap, H43, and Lemma A.1,

H100 $\vdash_i s; v; L^k[\text{trap}] : \mathcal{I}^*$

By Case C2, H3, H42, H44, H100, and the [Non-Root] rule, we conclude:

H101 $\vdash_i s; v; L^k[\text{trap}] : \mathcal{I}^*$

Case C2.7 is complete.

Case C2.8 $s; v; L^k[\text{prompt tf e end prompt\_end}] \rightarrow_i s'; v'; L^k[\text{block tf e end prompt\_end}] \land s' = \delta_p(s, i)$

By Case C2.8:

H102 $\delta_p(s, i) = s'$ where $s'$ is $s$ except $S'_{\text{inst}}(i)_{\text{pstack}}$ maps to $\text{push}(S_{\text{inst}}(i)_{\text{pstack}}, \text{etable} = \text{nil}', \text{root} = \text{nil})$

By H43, H102, the type checking of prompt, and Lemma A.1,

H103 $\vdash_i s'; v'; L^k[\text{block tf e end prompt\_end}] : \mathcal{I}^*$

By H3, H102 and store typing:

H104 $\vdash s'; S'$

H105 $p_r = \max\{p \mid S'_{\text{inst}}(i)_{\text{pstack}}(p)_{\text{root}} \neq \text{nil}\}$

H106 $S'_{\text{inst}}(i)_{\text{pstack}}(p_r)_{\text{root}} = \kappa_R$

H107 $S'_{\text{inst}}(i)_{\text{pstack}}(p_r)_{\text{table}}(\kappa_R) = t^*$

By H103, H104, H105, H106, H107, and the [Non-Root] rule:
Theorem A.3 (\(\rightsquigarrow\) Progress). If \(\vdash^k s; v^*; e^* : t^*\), then either \(e^* = v''\) or \(e^* = \text{trap}\) or \(s; v^*; e^* \rightsquigarrow s'; v''; e''\).

Proof: The reduction rules for control, restore, continuation\_copy, continuation\_delete, and prompt can be trivially checked to cover the space of possible well-typed configurations \(s; v^*; e^*\). Thus, all redexes for these instructions are guaranteed to take a step with \(\rightsquigarrow\) to a new configuration (possibly a step to a trap).

The non-trivial case is prompt\_end. Suppose \(\vdash^k s; v^*; L^k[\text{prompt\_end}] : t^*\). From the reduction rule of prompt\_end, a step \(s; v^*; L^k[\text{prompt\_end}] \rightsquigarrow s; v^*; L^k[\epsilon]\) will occur if getRoot(s, i) = nil.

However, no such step occurs if getRoot(s, i) ≠ nil. We thus want to show that getRoot(s, i) ≠ nil is in contradiction with \(\vdash^k s; v^*; L^k[\text{prompt\_end}] : t^*\), implying that this case cannot occur. By the [Prompt-End] rule and \(\vdash^k s; v^*; L^k[\text{prompt\_end}] : t^*\), we can deduce that \(\vdash s : S\) and \(S_{\text{inst}}(0)_{\text{Root}} = \text{nil}\). From this and the store typing rules, we find that getRoot(s, i) = nil, which is a contradiction.

\[\square\]