`Wasm/k`: WebAssembly Meets First-Class Continuations

Authors...
University of Massachusetts Amherst
United States

Abstract

TODO: Abstract

ACM Reference Format:
Authors.... 2020. `Wasm/k`: WebAssembly Meets First-Class Continuations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Contents

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction [NEED FEEDBACK]

For decades, JavaScript has been the only programming language executable by Web browsers without need of plugins, and has been an incredible source of growth for the modern Web. Web browsers have consolidated their power as the primary means of software delivery to consumers, and through this transformation consumers now expect more sophisticated applications than ever on the Web. Large programs such as spreadsheets, IDEs, and even video editors are available on the Web.

However, as the scale of Web applications grow, JavaScript appears to be more of a constraint than a catalyst. Large pieces of software may be more suitably written in languages other than JavaScript for performance or programmer productivity purposes. For example, a video editor may like to use an existing C library for encoding video, rather than implementing video encoding in JavaScript. One stop-gap solution has been to use JavaScript as a compilation target for various high-level languages. However, this approach has two key drawbacks: 1. performance remains poor, and 2. JavaScript lacks crucial features necessary for reliably implementing runtime features of high-level languages such as user-level threads.

WebAssembly [TODO: CITE] is a recently introduced low-level bytecode format which aims to replace JavaScript as a compilation target. WebAssembly is supported natively by all major Web browsers, and despite its name can also be used easily as a runtime environment outside of the Web. However, WebAssembly currently only addresses the drawback of performance mentioned above, and remains lacking in terms of allowing compilers targeting WebAssembly to easily implement important runtime features. For languages such as C/C++ which have minimal runtime features, WebAssembly serves as a viable compilation target, with the Emscripten compiler able to compile C/C++ code to WebAssembly with an average overhead of ___x compared to native code [TODO: CITE] . Unfortunately, a variety of languages, such as Go, support significant runtime features (user-level threads in Go's case), and cannot be compiled easily to WebAssembly. The compilation of these features increases the complexity of maintaining the compiler, and significantly impacts the performance of the generated code. Currently, WebAssembly is not a suitable compilation target for e.g. Go in performance-critical areas.

Limitations as a Compilation Target To understand why languages such as Go cannot be compiled in a straightforward manner to WebAssembly, we need a brief overview of the design of WebAssembly. One of the key design goals of WebAssembly is to be safe ([define safe here – Donald]). There are two broad categories of design choices which are crucial for the safety of WebAssembly. First, the state of the WebAssembly virtual machine is entirely opaque to executing WebAssembly code, and in particular the WebAssembly call stack is stored in a separate region of memory which is unable to be read or written by WebAssembly code. Second, WebAssembly allows control flow only through statically-checked structured nesting of blocks. That is, WebAssembly only allows jumps to lexically-scoped outer labels, rather than to arbitrary memory locations such as in x86 assembly.

The restrictions WebAssembly has in place to achieve safety inhibit the compilation of runtime features which either 1. read/write the virtual machine state and in particular the stack, or 2. jump execution non-locally. For example, user-level threads in Go are difficult to compile since thread context switching seemingly requires 1. switching to a new stack, and 2. jumping to an arbitrarily saved instruction address.

Current Workarounds To get around the above restrictions, the current Go compiler uses an duplicated shadow stack residing in user-memory to enable saving and switching stacks, and uses an elaborate state machine to emulate non-local jumps. The performance cost is substantial: up to ___x compared to native, and a code size increase of up to ___x compared to native as opposed to ___x and ___x respectively with C via Emscripten. Since the compilation strategy essentially affects code generation globally, this strategy forces programmers to pay a steep performance cost even with minimal usage of threads.

A related approach is the [TODO: CITE Asyncify] API, which extends WebAssembly with a stack manipulation API that allows for unwinding and rewinding stacks. This is accomplished via a source-to-source rewriting strategy very similar to Go's strategy. As with Go, this transformation adds a significant performance overhead, and is explored further in §6.

The difficulties already experienced in WebAssembly with compiling high-level language runtime features, combined with the related historic difficulties in JavaScript [TODO: CITE stopify] suggest a new approach should be considered.

Our Contributions In order to achieve our goal of enabling the compilation of high-level languages with diverse runtime features to performant WebAssembly, we explore extending WebAssembly with additional primitive instructions. The desired capabilities of 1. manipulating the stack(s), and 2. performing non-local jumps, hint at introducing *first-class continuations* into WebAssembly. In this paper we present `Wasm/k`, an extension which augments WebAssembly with full support for one-shot first-class continuations via a handful of new instructions.

First-class continuation operators are well-known abstractions allowing for powerful manipulations of the stack and control of the program. While first-class continuations have typically been implemented in high-level languages such as Racket and compiled to low-level languages without first-class continuations, in this work we invert this tradition and implement first-class continuations in the low-level setting of WebAssembly. In doing so, we develop adaptations to first-class continuations so as to carefully maintain the safety of WebAssembly, in particular in the case of heterogenous call stacks caused by foreign language interoperation.

In designing our extension, we wanted to ensure the new primitives fit well in WebAssembly, and align with WebAssembly's performance, portability, and safety objectives. We considered the following design goals: 1. Common runtime features, such as user-level threads, should be able to compile to efficient `Wasm/k` code; 2. The extension should be simple, so as to keep the formal semantics and type checking tractable to reason about, as well as maintaining fast parsing and type checking speeds in practice; 3. Safety must be maintained; 4. Existing WebAssembly instructions and code should suffer no performance penalty; 5. The performance of each new instruction should be both fast and predictable.

In order to achieve these design goals, we offer the following contributions:

1. To enable readable and rapid prototyping of code examples we develop `C/k`, an extension of C with first-class continuations which compiles to `Wasm/k`, and explore a variety of examples in `C/k` as a means of informally introducing the `Wasm/k` API (§2).
2. The formal semantics and type checking of `Wasm/k`, formulated as a simple extension of WebAssembly (§3).
3. Proofs showing that the desired safety properties of `Wasm/k` hold in a formal setting (§3.3).
4. A detailed informal extension of the proofs arguing why desired safety properties hold in the practical situation of inter-language function calls. In order to achieve such safety, prompts (delimited continuations) [TODO: CITE] turn out to be necessary. (§4).
5. An implementation of `Wasm/k` in Wasmtime, an efficient real-world WebAssembly JIT (§5).

We evaluate the performance of `Wasm/k` in a series of microbenchmarks in §6.

2 A Tour of First-Class Continuations in WebAssembly

We start by analyzing a sequence of examples, which will first introduce WebAssembly for unfamiliar readers and provide valuable context for understanding compilation to WebAssembly to motivate `Wasm/k`, and then introduce the core instructions of `Wasm/k`: (**control** h), **restore**, **continuation_copy**,

```
1 func foo(n int64) float64 {
2   f := float64(0)
3   for k := int64(1); k <= n; k++ {
4     f += 1.0 / float64(k)
5     bar() // thread switch???
6   }
7   return f
8 }
```

(a) Example Go program

```
1 (local.set $f (f64.add
2   (f64.load offset=8 (local.get $sp))
3   (f64.div
4     (f64.const 1)
5     (f64.convert_i64_s
6       (i64.load (local.get $sp)))))))
```

```
1 (i64.store (local.get $sp)
2   (local.get $k))
3 (f64.store offset=8 (local.get $sp)
4   (local.get $f))
5 (global.set $gsp
6   (local.tee $sp (i32.sub
7     (local.get $sp)
8     (i32.const 8))))
9 (i64.store (local.get $sp)
10  (i64.const 383975427))
11 (call $bar (i32.const 0))
12 (local.set $sp (global.get $gsp))
```

(b) Code generated by Go

```
1 (local.set $f (f64.add
2   (local.get $f)
3   (f64.div
4     (f64.const 1)
5     (f64.convert_i64_s (local.get $k)))))
```

```
1 (call $bar)
```

(c) Code generated by Emscripten for equivalent C code

and **continuation_delete**. These four instructions can be used to implement features as diverse as user-level threads and probabilistic programming, as will be shown by increasingly advanced examples.

2.1 Case Study of Code Generation

Figure 1a shows an example of a simple function `foo` in Go. The function computes a partial sum of a series, and upon each iteration calls the function `bar`, which in this example we assume the compiler chooses not to inline. The Go compiler assumes that the function `bar` may include code which triggers a thread switch in Go, such as a blocking call, and thus generates code for `foo` accordingly. While we omit the full code generation for the sake of brevity, we show instructive portions of the code generation.

The code generated by Go for lines 4 and 5 of `foo` is shown in the top and bottom of Figure 1b, respectively. The generated code exhibits the two primary methods for storing data in WebAssembly. First, data can be stored in *local variables* via **local.get** / **local.set** which are conceptually similar to local variables in C: formally they live in the WebAssembly stack and practically they will be placed in registers during JIT compilation. Second, data can be stored in *linear memory* via e.g. **f64.load** / **f64.store**. In Figure 1b the loads and stores have been underlined. Since the WebAssembly stack is inaccessible by WebAssembly code, the Go compiler is forced to create a copy of the stack residing in linear memory, a so called *shadow stack*, and upon thread switching the shadow stacks in linear memory can easily be switched.

In the top of Figure 1b, we can see that the current values of `f` and `k` are retrieved from the shadow stack in linear memory via load instructions, and a local variable copy of `f` is updated. Conversely, the lines 1–4 of the bottom half of Figure 1b show that the local copies of `f` and `k` must be saved into the shadow stack in linear memory prior to the call to `bar`. This ensures that if a thread switch occurs in `bar`, the values of `f` and `k` will be properly saved. Asyncify generates similar code which must save local variables into memory before function calls, and load them back from memory during use.

Since C does not offer the runtime feature of user-level threads, the Emscripten compiler is able to generate simpler code than Go. Figure 1c shows the code which Emscripten generates for equivalent C code. No loads or stores are emitted by Emscripten, and WebAssembly local variables are used exclusively to store `f` and `k`. During JIT compilation, both of these WebAssembly locals will be allocated to registers.

We can quantify the additional instructions introduced by Go compared to Emscripten. In a call of `foo(`2^{30}`)`, the `perf` tool shows that executing under `node v14.4.0`, the Go program as compared to the C program executes $\approx 2.5x$ more instructions, $\approx 3x$ more branches, $\approx 1.9x$ more loads, and $\approx 1.5x$ more stores. Overall, the execution time of the Go program takes $\approx 1.77x$ longer than the C program. We do not intend for this comparison to regard Go and C as competitors, but rather to illustrate the potential gain in performance if in the future the Go compiler could generate code more similar to Emscripten's code by making use of `Wasm/k`.

2.2 A Wasm Quadrupling Function, Served Two Ways

The Basics of WebAssembly Wasm code is contained in *modules*, which consist of multiple functions, among other possible pieces of data such as global variables and indexable memory. Wasm is a stack machine, so each instruction of Wasm modifies the values currently on the *value stack* in some manner: a (**i64.const** 2) instruction pushes a 2 onto the value stack, while a **i64.mul** instruction pops two values, and then pushes the multiplication result. Function calls (**call** $\$f$) work by popping the arguments off the value stack, executing the function, and pushing the return value on the value stack. Finally, a function can read the value of an

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

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

```
1 (module
2    (func $handler (param $k i64) (param $x i64)
3       local.get $k
4       local.get $x
5       i64.const 2
6       i64.mul
7       restore)
8    (func $quadruple2 (param $x i64) (result i64)
9       local.get $x
10      control $handler
11      i64.const 2
12      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.

argument (or local variable) by (**local**.**get** $*x*) which pushes the argument value onto the value stack. For example, the code in Figure 2a implements a function `$quadruple` which quadruples its argument by first calling a function `$helper` which doubles the argument, and the doubling again the return value of the argument.

A Simple Use of Continuations A trivial use of first-class continuations is to mimic the behavior of function calls. Figure 2b shows an alternative way to write the `$quadruple` function, this time making use of the (**control** $*h*) and **restore** instructions that we have added to Wasm. At line 10, `$quadruple2` instead executes `$handler` inside an entirely new and empty call stack. Two arguments are passed to `$handler`:

1. `$k` is the *captured continuation* at the point of **control**. This represents the execution that remains to be done after the **control**, and includes a saved version of the call stack. The value of `$k` is generated automatically by the **control** instruction.
2. `$x` is simply the arbitrary user-provided argument, similar to the argument in an ordinary function call.

Next, at line 7, `$handler` performs a **restore**, with `$k` and a doubled value as arguments. The **restore** instruction effectively undoes the **control** operator: **restore** will jump back to the continuation saved in `$k` passing the doubled value back, and deallocate the fresh call stack that `$handler` was

```
1 (module
2    (func $enqueue (param $k i64) <code omitted...>)
3    (func $dequeue (result i64) <code omitted...>)
4    (func $handler (param $k i64) (param $unused i64)
5       local.get $k
6       call $enqueue
7       call $dequeue
8       i64.const 0
9       restore)
10   (func $sleep
11      i64.const 0
12      control $handler
13      drop))
```

Figure 3. Implementing user-level threads via first-class continuations in Wasm

executing in. At this point, line 11 will execute, with a doubled valued pushed onto the stack. Execution completes as before by doubling again.

2.3 User-Level Threads

The example of Figure 2 is a minimal example meant to introduce the basic ideas of **control** and **restore**. A more substantial use of first-class continuations is to implement user-level cooperative threads in Wasm. The goal is for a thread to be able to call a `$sleep` function which will pause the current thread and find a different thread to resume.

The key insight is that pausing a thread can be accomplished by capturing the current continuation of the thread via **control**, and resuming another thread corresponds to **restore**-ing the captured continuation of the desired thread. The core of the idea is shown in Figure 3, which first uses **control** to capture the current continuation `$k` and save it in the waiting queue. Then another continuation is dequeued and **restore**-d. Note that since `$sleep` only wants to capture and restore the program state and not pass data between threads, the arguments to **control** and **restore** are unused in this example. We will return to this example in §6.2 to evaluate the performance of this implementation.

2.4 Exposing First-Class Continuations to C

[[[TODO: Possibly move this to before the user-threads, and then show the user-threads in C/C++ with an example]]]

Writing and analyzing substantial examples directly in Wasm is somewhat tedious. The implementations of `$enqueue` and `$dequeue` above were omitted since they are rather long in Wasm. Fortunately, we can alleviate this burden by using C/C++. Since C/C++ compiles directly to Wasm via the Emscripten compiler, and we have added first-class continuations as primitives to Wasm, we can now add first-class continuations to C/C++. Concretely, we have defined the function prototypes shown in Figure 4, and wrapped the Emscripten compiler such that calls to these functions will be compiled directly to the corresponding Wasm instruction. Note that the functions (and Wasm instructions)

```
uint64_t control(uint64_t arg, control_handler_fn fn_ptr);
void restore(uint64_t k, uint64_t val);
uint64_t continuation_copy(uint64_t k);
void continuation_delete(uint64_t k);
```

Figure 4. A C/C++ First-Class Continuations Header

continuation_copy and continuation_delete have not yet been explained, but they will be explored soon in the following examples.

The very simple translation that our Emscripten wrapper performs of compiling control in C/C++ directly to **control** in WebAssembly (and likewise for the other instructions) unfortunately introduces some important caveats. With this technique, we are only able to support a subset of C and C++, specifically the subset which does not require stack frames to be placed on a shadow stack in linear memory. In particular, pointers to values on the C stack, passing C++ objects by reference rather than by pointer, and C++ exceptions are not supported. Fortunately, these restrictions are not fundamental, but do require substantially more engineering work at the compilation stage. See §4.2 for a discussion of solutions to this problem. [Arjun: not sure if its ok to reference a future section like this? – Donald] Despite these restrictions, we are able to make use of a large subset of C++, including the standard library.

Adding first-class continuations in this manner to 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. In this case we are going in the opposite direction: we have added first-class continuations to a low-level language, and now get them almost for free in a higher-level language.

2.5 Generators

A common abstraction in a variety of languages (such as Python) is the concept of a *generator*. A generator is a function which can "yield" a value at any point, which pauses the execution of the generator and delivers the value to the caller of the generator. The caller can then call the generator again to resume execution of the generator. While C does not have generators, we can implement generators in terms of control and restore.

As an example of what we would like to support for generators, consider the example generator usage in the top half of Figure 5. Execution starts in main, and upon executing gen_next on line 10, execution will jump to example_generator. Once example_generator executes gen_yield on line 4, execution will jump back to the printf of line 10, with 0 being the "return" value of gen_next. This will repeat with example_generator calling gen_yield with incrementing values, until the for loop completes.

To implement this API of generators, we can start by thinking about what program states will need to be saved in continuations. When gen_yield is called, the state (i.e. the stack) of example_generator needs to be saved. Conversely, gen_next needs to save the stack of main. We add fields for each of these continuations in the Generator struct, as well as a temporary field need to hold a value (bottom half of Figure 5, line 3).

Lines 4–13 implement a helper function convertFuncToCont which takes as input a function pointer, and returns a continuation such that when the continuation is restored, the given function pointer will be called. Nothing deeply new

```
1  void example_generator(Generator *g) {
2      uint64_t i = 0;
3      while(1) { gen_yield(i++, g); }
4  }
5  int main() {
6      Generator *g = make_generator(example_generator);
7      for(int i = 0; i < 10; i++)
8          printf("%llu\n", gen_next(g));
9      free_generator(g);
10     return 0;
11 }
```

```
1  typedef struct {
2      k_id after_next, after_yield; uint64_t value;
3  } Generator;
4  // Helpers for converting a function to a continuation
5  void return_convert_result(uint64_t k, uint64_t ak) {
6      restore(ak, k);
7  }
8  void convert_handler(uint64_t k, void (*f)(Generator*)) {
9      f((Generator *)control(return_convert_result, k));
10 }
11 uint64_t convertFuncToCont(void (*f)(Generator*)) {
12     return control(convert_handler, f);
13 }
14 // Allocating a generator
15 Generator *make_generator(void (*f)(Generator*)) {
16     Generator *g = (Generator *)malloc(sizeof(Generator));
17     g->after_yield = convertFuncToCont(f); return g;
18 }
19 // Yielding implementation
20 void yield_handler(k_id k, Generator *g) {
21     g->after_yield = k;
22     restore(g->after_next, g->value);
23 }
24 void gen_yield(uint64_t v, Generator *g) {
25     g->value = v;
26     control(yield_handler, g);
27 }
28 // Next implementation
29 void next_handler(k_id k, Generator *g) {
30     g->after_next = k;
31     restore(g->after_yield, 0);
32 }
33 uint64_t gen_next(Generator *g) {
34     return control(next_handler, g);
35 }
36 // Freeing a generator
37 void free_generator(Generator *g) {
38     continuation_delete(g->after_yield); free(g);
39 }
```

Figure 5. Example usage and implementation of generators.

is presented in lines 4–9, but it is a good exercise to understand how it works. The helper function is used on line 17 to initialize a Generator for a given function pointer.

The core of the implementation is in the functions gen_yield and gen_next (lines 19–35). Suppose that currently example_generator is executing, and calls gen_yield. Conceptually, we want to save the execution state of example_generator and then switch over to the execution state of main, which must have been saved previously. Saving the execution state of example_generator is done using control, and then switching back to main is done using restore. gen_next works almost identically in reverse.

Finally, we need to be able to free our Generator. Consider the final call to gen_next. After gen_next, example_generator is executed, and calls gen_yield for the last time. In the final gen_yield, the continuation of example_generator is saved, and then main is restored to. At this point main is done and no longer calls gen_next, and hence the continuation of example_generator that was just captured will never be consumed (i.e. restored to). If we do not free this continuation somehow, it will be leaked. Therefore, we need to have an instruction for freeing a continuation, which is what continuation_delete does on line 38.

The need for a **continuation_delete** instruction is subtle, and reflects the reality of implementing first-class continuations in a low-level assembly language. To compare with Racket again, a high-level language will typically feature a garbage collector, and continuations will be garbage collected as any other value when no longer referenced.

2.6 Probabilistic Programming

A more involved example is the implementation of an embedded probabilistic programming language in C++. [[[TODO: blurb about what are PPLs and why they are cool]]] . Previous pieces of work by [TODO: CITE https://arxiv.org/pdf/1403.0504.pdf] and [TODO: CITE http://proceedings.mlr.press/v84/ge18b/ge18b.pdf] have shown promise in using control operators to implement probabilistic programming languages efficiently. In particular, [TODO: CITE https://arxiv.org/pdf/1403.0504.pdf] implements a probabilistic programming language embedded in C by making use of operating system level control abstractions such as fork, and [TODO: CITE http://proceedings.mlr.press/v84/ge18b/ge18b.pdf] make use of Julia's coroutines to implement an embedded probabilistic programming language. Fundamentally, these approaches work by relating the sampling from a probability distribution to sampling from a distribution of program executions. 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++.

```
1 uint64_t sum_d6() {
2     auto *d6 = new std::vector<uint64_t> {1, 2, 3, 4, 5, 6};
3     return uniform(d6) + uniform(d6);
4 }
5 int main() {
6     std::cout << *driver(sum_d6) << std::endl; return 0;
7 }
```

```
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 }
```

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

An example usage of the embedded probabilistic programming language is shown in the top half of Figure 6. sum_d6 computes the sum of two independent dice rolls. The call to driver(sum_d6) will run the sampling algorithm, eventually returning a std::map<uint64_t, double> representing the 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 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 the bottom half of Figure 6. 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 1–5), 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 7–15) 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 12) will jump back into the execution of somewhere in body, eventually returning yet again to the `push_back` (line 9). Thus, `driver` will continue to push results and dequeue a new thunk, until all thunks (samples) are exhausted. Finally, `count_probs` simply computes the desired `std::map<uint64_t, double>`.

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 24) 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 19). 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 21).

3 A Formalized Semantics (section name could be better)

[This is repetitive with the last paragraph. What's the right way to fix this? – Donald] In order to describe a formalized semantics of our four new first-class continuation instructions, we start by giving an overview of the original semantics of Wasm. We then describe the semantics of our instructions, and finally discuss subtleties of our design.

3.1 Overview of Wasm Semantics

The formal semantics of Wasm are defined using a stack-based small-step reduction semantics. We will describe various aspects of the semantics while walking through the example Wasm code shown in Figure 7.

The Wasm Stack Wasm is specified as a stack machine, in which the stack heterogeneously stores both instructions to be evaluated as well as evaluation results. Indeed, a fully-evaluated value is represented as a **const** instruction, such as (**i64.const** 3) in Figure 7. In the absence of control flow and function calls, the Wasm stack will always have the shape v^*e^*, where v^* contains values, i.e. **const** instructions, and e^* contains yet to be evaluated instructions. Evaluation proceeds by evaluating instructions at the boundary of the values and instructions. For example, in Figure 7, ignoring the **label** and **end** for now, the next instruction to be evaluated is the **i64.add**, acting on the values (**i64.const** 3) and (**i64.const** 4). The boundary between values and instructions at which evaluation occurs is called the *local context of depth 0* ($L^0[\_]$).

Wasm Control Flow Control flow in Wasm is *structured*, that is, rather than consisting of jumps to arbitrary labels, control flow is specified by delimited control flow blocks given by $\textbf{label}\{e^*\}$ and **end** instructions, where e^* indicates additional instructions to evaluate after jumping to the given label. Note that **label** is not a user-accessible instruction, but is an administrative instruction which user-accessible control flow constructs are defined in terms of. These control flow blocks must be properly nested, as shown with the paired $\textbf{label}\{\epsilon\}$ and **end** in Figure 7. Evaluation proceeds by executing the instructions inside the most deeply nested control blocks.

The nesting of control blocks is formalized 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 control block. For example, $L^1[(\textbf{i64.const}\ 3)\ (\textbf{i64.const}\ 4)\ \textbf{i64.add}]$ matches the stack in Figure 7, but $L^0[(\textbf{i64.const}\ 3)\ (\textbf{i64.const}\ 4)\ \textbf{i64.add}]$ does not.

Stores and Instances The execution of various Wasm instructions requires accessing global state. For instance, function calls require indexing in the immutable table of functions, and memory operations read and write the global linear memory. The mutable global state of the execution is maintained in the *store* (s).

The Reduction Rules of Wasm The Wasm stack, nested control flow, and stores make up the interesting components of the semantics of Wasm. With these defined, the semantics of Wasm is then defined as a small-step semantics given by a reduction relation $\hookrightarrow$ on the instructions e^*, and possibly reading or writing s. The evaluation rules of Wasm are defined in the form $s; e^* \hookrightarrow s'; e'^*$, and are defined to be congruent with local contexts, such that if $s; e^* \hookrightarrow s'; e'^*$ then $s; L^k[e^*] \hookrightarrow s'; L^k[e'^*]$, meaning that evaluation always occurs in the most deeply nested local context, unless no such evaluation is possible. During reductions in which the store is irrelevant the store is left implicit. For example, the execution trace of Figure 7 is:

$$\begin{aligned} & 2\ \textbf{label}\{\epsilon\}\ 3\ 4\ \textbf{i64.add}\ (\textbf{br}\ 0)\ \textbf{end}\ \textbf{i64.sub} \\ \hookrightarrow\ & 2\ \textbf{label}\{\epsilon\}\ 7\ (\textbf{br}\ 0)\ \textbf{end}\ \textbf{i64.sub} \\ \hookrightarrow\ & 2\ 7\ \textbf{i64.sub} \\ \hookrightarrow\ & (-5) \end{aligned}$$

where an integer x is shorthand for the WebAssembly value (**i64.const** x).

(**i64.const** 2) **label**{ϵ} (**i64.const** 3) (**i64.const** 4) **i64.add** (**br** 0) **end** **i64.sub**

Figure 7. An example Wasm stack.

3.2 Semantics of One-Shot Continuations

To develop the formal semantics of first-class continuations in Wasm, we first describe the new values, types and instructions, then we consider the additions made to the runtime structure of Wasm (the store), and finally we describe the extensions to the reduction rules.

New values, types, and instructions First-class continuations are typically featured in high-level languages such as Racket. In such high-level languages, the captured continuation (κ) is usually represented as a function value in the language, which when applied to an argument restores the captured continuation. However, Wasm does not support first-class functions, so representing a captured continuation (κ) as a Wasm function is not a viable choice. Instead, we introduce a new type $\boldsymbol{\kappa}$**_*id*** which is the type inhabited by *continuation IDs*, which will uniquely identify live continuations. In the semantics that follow and in our implementation, we set $\boldsymbol{\kappa}$**_*id*** ::= **i64** and thus we do not add it as an explicit type to Wasm. [1] The new values that we add to Wasm are precisely the continuation IDs. These continuation IDs can be manipulated by existing instructions such as $\boldsymbol{\kappa}$**_id.store** or $\boldsymbol{\kappa}$**_id.load**, or they can be manipulated by four new instructions:

1. (**control** f): Captures the current continuation and executes the function f with the ID of the captured continuation (κ) as an argument. An additional arbitrary **i64** argument is also passed, to support the extremely common case of a closure environment being passed to f.
2. **restore**: Restores the continuation of a given continuation ID, consuming the ID in the process.
3. **continuation_copy**: Explicitly makes a copy of the continuation of a given continuation ID, returning a new continuation ID.
4. **continuation_delete**: Explicitly deletes the continuation of the given continuation ID.

Additions to the Runtime Structure One of the crucial features of Wasm is safety, which ensures that Wasm programs can only go wrong in a limited number of ways. In particular, malicious code can not corrupt the control stack of Wasm. In order to preserve the safety of Wasm, it is crucial to ensure that captured continuations are saved in memory such that they are not directly writable by Wasm code, as otherwise malicious Wasm code could write arbitrary instructions into a captured continuation, and then jump (**restore**) to that continuation.

Therefore, we do not save captured continuations in Wasm's existing linear memory, which allows the programmer direct read-write access. Instead, we save captured continuations in a new region of memory, the *continuation table*. Wasm code is not allowed to directly index into the table. Rather, the four new instructions will operate on the table on behalf of the programmer.

During execution the store (s) now contains an array of continuation tables, and each instantiated Wasm module (*inst*) possibly contains an index into that array of tables. A continuation table (*ctableinst*) contains an array of entries, where each entry is either *nil* or a captured continuation, with all entries initialized to *nil*. The table is finite, which puts an upper limit on the number of continuations which can be simultaneously stored. Like stack overflows, the exact size of the table depends on the Wasm runtime environment. For convenience, we define the administrative function cont(s, i, κ) to index into the continuation table for instantiated Wasm module i at continuation ID κ.

A captured continuation is saved in the continuation table as a record containing the values of local variables and the entire evaluation context at the time of capture. However, representing the entire evaluation context at the time of capture is a bit tricky since the control block depth of the stack at the time of capture is not fixed, but a local context (L^k) demands a fixed depth. We define a *full-stack context* ($L^{\max}$) as an evaluation context similar to L^k, but it can match a stack with arbitrary control block depth. The saved continuation stores one of these matched full-stack contexts.

Finally, the store also needs to save the index of the *root continuation*, that is, the continuation associated with the stack which initialized the Wasm execution. 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. The necessity of this runtime information will be explained further in §4.1 [I know Arjun said I shouldn't do this. Need to avoid somehow – Donald]. The administrative function root(s, i) is defined to return the root ID or nil for the instance i. All of these additions to the runtime structure of Wasm are shown in the top part of Figure 8.

Extensions to the Reduction Rules The original semantics of Wasm define the reduction relation $\hookrightarrow$, and define it to be congruent with local contexts. This congruence reduction rule is reproduced verbatim as the [Cong] rule in Figure 8. However, full congruence with local contexts is no longer correct in the presence of first-class continuations.

[1] One could reasonably use **i32** or keep $\boldsymbol{\kappa}$**_*id*** as a distinct type, in which case additional nearly-duplicate instructions such as $\boldsymbol{\kappa}$**_id.load** would have to be added to support storage in linear memory.

$$
\begin{aligned}
\boldsymbol{\kappa\_id} &::= \textbf{i64}\\
e &::= \cdots \mid \textbf{control}\ f \mid \textbf{restore} \mid \textbf{continuation\_copy} \mid \textbf{continuation\_delete}\\
L^{\max} &::= v^*[\_]e^* \mid v^*\ \textbf{label}_n\{e^*\}\ L^{\max}\ \textbf{end}\ e^*\\
s &::= \{\text{inst } inst^*, \text{tab } tabinst^*, \text{mem } meminst^*, \text{pstacks } pstack^*\}\\
inst &::= \{\text{func } cl^*, \text{glob } v^*, \text{tab } i^?, \text{mem } i^?, \text{pstack } i^?\}\\
pstack &::= pinst^*\\
pinst &::= \{\text{ctable } (\{\text{locals } v^*, \text{ctx } L^{\max}\} \mid nil)^*, \text{root } (\boldsymbol{\kappa\_id} \mid nil)\}
\end{aligned}
$$

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

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

$$
\begin{array}{rll}
s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ v)\ (\textbf{control}\ j)] \rightsquigarrow_i & & \\
s';\ \epsilon;\ (\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ (\textbf{call}\ j)\ \textbf{trap} & & \text{if } \delta_{\text{ctrl}}(s, s', i, \kappa, v_l^*, L^{\max})\\
s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\textbf{i64.const}\ v)] & & \text{if } \delta_{\text{rest}}(s, s', i, \kappa, v_l^*, L^{\max})\\
s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i s;\ v_l^*;\ \textbf{trap} & & \text{otherwise}\\
s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_copy} \hookrightarrow_i s';\ (\textbf{i64.const}\ \kappa') & & \text{if } \delta_{\text{copy}}(s, s', i, \kappa, \kappa')\\
s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_copy} \hookrightarrow_i s;\ \textbf{trap} & & \text{otherwise}\\
s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_delete} \hookrightarrow_i s';\ \epsilon & & \text{if } \delta_{\text{delete}}(s, s', i, \kappa)\\
s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_delete} \hookrightarrow_i s;\ \textbf{trap} & & \text{otherwise}\\
s;\ \textbf{prompt}\ tf\, e^*\ \textbf{end} \hookrightarrow_i s';\ \textbf{block}\ e^*\ \textbf{end}\ \textbf{prompt\_end} & & \text{if } \delta_{\text{p}}(s, s', i)\\
s;\ \textbf{prompt\_end} \hookrightarrow_i s';\ \epsilon & & \text{if } \delta_{\text{p-end}}(s, s', i)
\end{array}
$$

[These are transition functions, describing how stores s are updated – Donald]

$$
\begin{aligned}
\delta_{\text{ctrl}}(s, s', i, \kappa, v_l^*, L^{\max}) &::= \text{cont}(s, i, \kappa) = nil \wedge \text{cont}(s', i, \kappa) = \{\text{locals} = v_l^*, \text{ctx} = L^{\max}\}\\
&\quad \wedge [(\text{root}(s, i) = nil \wedge \text{root}(s', i) = \kappa) \vee (\text{root}(s, i) = \kappa_R \wedge \text{root}(s', i) = \kappa_R)]\\
\delta_{\text{rest}}(s, s', i, \kappa, v_l^*, L^{\max\prime}) &::= \text{cont}(s, i, \kappa) = \{\text{locals} = v_l^{*\prime}, \text{ctx} = L^{\max\prime}\} \wedge \text{cont}(s', i, \kappa) = nil\\
&\quad \wedge \text{root}(s, i) = \kappa_R \wedge ((\kappa = \kappa_R \wedge \text{root}(s', i) = nil) \vee (\kappa \neq \kappa_R \wedge \text{root}(s', i) = \kappa_R))\\
\delta_{\text{copy}}(s, s', i, \kappa, \kappa') &::= \text{cont}(s, i, \kappa) \neq nil \wedge \text{cont}(s, i, \kappa') = nil \wedge \text{cont}(s', i, \kappa') = \text{cont}(s, i, \kappa) \wedge \text{root}(s, i) \neq \kappa\\
\delta_{\text{delete}}(s, s', i, \kappa) &::= \text{cont}(s, i, \kappa) \neq nil \wedge \text{cont}(s', i, \kappa) = nil \wedge \text{root}(s, i) \neq \kappa\\
\delta_{\text{p}}(s, s', i) &::= s'_{\text{pstacks}}(s'_{\text{inst}}(i)_{\text{pstack}}) = \text{push}(s_{\text{pstacks}}(s_{\text{inst}}(i)_{\text{pstack}}), \{\text{ctable} = nil^*, \text{root} = nil\})\\
\delta_{\text{p-end}}(s, s', i) &::= s'_{\text{pstacks}}(s'_{\text{inst}}(i)_{\text{pstack}}) = \text{pop}(s_{\text{pstacks}}(s_{\text{inst}}(i)_{\text{pstack}}))
\end{aligned}
$$

$$
\begin{aligned}
\text{root}(s, i) &::= \text{top}(s_{\text{pstacks}}(s_{\text{inst}}(i)_{\text{pstack}}))_{\text{root}}\\
\text{cont}(s, i, \kappa) &::= \text{top}(s_{\text{pstacks}}(s_{\text{inst}}(i)_{\text{pstack}}))_{\text{ctable}}(\kappa)
\end{aligned}
$$

Figure 8. Additional semantics reduction rules.

For example, the following reduction is correct:

$$s;\ (\textbf{i64.const}\ 0)\ (\textbf{control}\ h)\ (\textbf{i64.const}\ 5)\ \textbf{i64.add}$$
$$\hookrightarrow s';\ \langle\texttt{code of h}\rangle$$

where s' has saved the continuation $[\_]$ ($\textbf{i64.const}$ 5) $\textbf{i64.add}$. However, the following reduction is *not* the correct semantics of first-class continuations, despite it being just the reduction above applied to the local context $L^0 = (\textbf{i64.const}\ 3)\ [\_]$:

$$s;\ (\textbf{i64.const}\ 3)\ (\textbf{i64.const}\ 0)\ (\textbf{control}\ h)\ (\textbf{i64.const}\ 5)\ \textbf{i64.add}$$
$$\hookrightarrow s';\ (\textbf{i64.const}\ 3)\ \langle\texttt{code of h}\rangle$$

where s' has saved the continuation $[\_]$ ($\textbf{i64.const}$ 5) $\textbf{i64.add}$. Therefore, in our extended semantics we only want to have partial congruence with local contexts, such that congruence is defined for reductions that do not involve the (**control** h) or **restore** instructions. Figure 8 defines a new reduction relation $\rightsquigarrow$ representing reductions which may involve use of (**control** h) or **restore**. This new reduction relation is now the primary relation describing the execution of Wasm in our extension. The relation $\hookrightarrow$ still refers exactly to the original Wasm reduction relation. Congruence is retained for $\hookrightarrow$ via the [Cong] rule, but there is no such congruence 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.

With the reduction relation $\rightsquigarrow$ introduced, we can now move on to describe the reduction rules of the four new instructions. Each instruction has two rules associated with it, one for the successful case and the other for the error case (resulting in a **trap**).

[[[TODO: Need to describe the root stacks stuff]]]

(1a) To evaluate a (**control** h) instruction with a single control argument (v), first the Wasm runtime chooses an unused continuation ID (κ). Second, the entire stack ($L^{\max}$) other than the control argument is captured and saved in the store (s) at the continuation ID (κ), along with the current local variable values (v_l^*). Evaluation then proceeds by calling the control handler function (h) in an empty stack with the continuation ID (κ) and control argument (v) as arguments, followed by a **trap**, which is an uncatchable exception in Wasm. It is a programmer error to allow a control handler to finish executing. If this were to occur, the **trap** would be triggered.

(1b) A (**control** h) instruction may **trap** if the Wasm runtime environment has no more space to allocate in the continuation table. Just as calling functions many times without returning will cause a stack overflow, invoking control handlers many times without restoring will cause the continuation table to be full.

(2a) Evaluating a **restore** instruction with two arguments: a continuation ID (κ) and a restore value (v) requires looking up the saved continuation at the given continuation ID (κ). The saved continuation is a record containing saved local variable values ($v_l^{*\prime}$) and the saved stack ($L^{\max\prime}$). Next, the saved local variables ($v_l^{*\prime}$) are restored, and the saved stack ($L^{\max\prime}$) is restored by substituting the restore value (v) into the hole of the saved stack ($L^{\max\prime}[(\textbf{i64.const}\ v)]$). Finally, the continuation is consumed, by marking the continuation ID (κ) as *nil* in the continuation table. In the future the same continuation ID could be chosen by (**control** h).

(2b) It is invalid to invoke **restore** on a continuation ID (κ) that is un-allocated. In this case, a **trap** will occur.

(3a) A **continuation_copy** instruction performs only local stack operations. Thus, **continuation_copy** is defined using the $\hookrightarrow$ reduction relation. To evaluate a **continuation_copy** instruction with a continuation ID (κ) as an argument, first an unused continuation ID (κ') is chosen, just as in the evaluation of (**control** h). Then the saved continuation at the given continuation ID (κ) is copied into the new continuation ID (κ'). The new continuation ID is then returned.

(3b) Like (**control** h), **continuation_copy** may **trap** if no space is left in the continuation table. In addition, a **trap** occurs if the provided continuation ID (κ) does not reference an alive continuation (the continuation table entry is *nil*).

(4a) To evaluate a **continuation_delete** instruction, the given continuation ID (κ) is simply deallocated (marked as *nil* in the continuation table).

(4b) It is invalid to attempt to delete a continuation which is not allocated, and results in a **trap**.

In addition, each new instruction reads and possibly updates the *root* continuation ID, to ensure at runtime safe usage of first-class continuations. The *root* continuation is initialized to *nil*, meaning that computation starts with the active continuation being the root continuation. The (**control** h) instruction sets the root continuation ID if (**control** h) is invoked from within the root continuation, and otherwise leaves it unmodified. Inversely, **restore** will set the root continuation ID back to *nil* or leave it untouched, depending on if the root continuation is being restored or not. Finally, **continuation_copy** and **continuation_delete** disallow copying or deleting the root continuation (**restore** also disallows implicitly deleting the root continuation). Disallowing copying and deleting the root continuation is required to guarantee that first-class continuations in Wasm do not compromise the safety of the host environment: the root continuation will typically include stack frames of the host environment, and copying or deleting these stack frames is dangerous. These concerns are explored further in §4.1.

$$C ::= \{\ldots, \text{label } ((t^*)^*)^*\}$$

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

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

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

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

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

Figure 9. Type Checking First-Class Continuations in Wasm.

3.3 Validation and Soundness [NEED FEEDBACK]

WebAssembly provides a provably sound environment in which untrusted code can be executed efficiently and safely. WebAssembly achieves this safety by performing validation via type checking on the input program, before execution. A necessary requirement of Wasm/k is to ensure that the safety of WebAssembly is maintained with the addition of first-class continuations. First we explain the required additions to the type checking, and then discuss how soundness properties extend to first-class continuations.

Typing Rules for One-Shot Continuations Type checking is performed in a stack-based syntax-directed manner. Most instructions have an associated type $t_1^* \longrightarrow t_2^*$, which indicates that the instruction consumes values of types t_1^* and produces values of types t_2^*. The type checking of the five new instructions fits easily into this framework, since unlike the semantics of the new instructions, the type checking of the new instructions acts only locally on the stack. Only one modification is made to the data structure of the context (C), which will be explained shortly when discussing the type checking of **prompt**.

Semantically, the (**control** h) instruction first consumes an **i64** value and then switches to another stack. If the original stack is restored, then an **i64** value will be pushed on. Thus, (**control** h) has type **i64** $\longrightarrow$ **i64**, subject to the constraint that the function h has the proper type of a control handler (**i64 i64** $\longrightarrow \epsilon$). The type checking of **restore** is similar to the **trap** instruction, as both instructions guarantee that instructions after them are unreachable. Thus, **restore** should likewise accept an arbitrary stack shape and produce an arbitrary stack shape. However, **restore** also needs to check that it has two **i64** arguments. The **continuation_copy** and **continuation_delete** instructions are straightforward: both consume a single **i64** and produce an **i64** and ϵ, respectively.

Type checking the **prompt** instruction is more involved, since we need to ensure that the semantics behave as expected. 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 which jump to labels lexically outside of the **prompt**. It would be incorrect to allow such a branch instruction to jump beyond the end of the **prompt** and thereby skip the required tearing-down of the prompt environment. To remedy this, we could modify the semantics to force the **prompt_end** instruction to be run even when branching past it. However, this does not fit clearly into the existing WebAssembly semantics, and would seemingly entail significant effort both formally with semantics and proofs, and at the implementation level with non-trivial consequences for code generation.

A simpler approach is to use the type checker to outlaw branching instructions which jump beyond the scope of the **prompt** (though still allow branches within the **prompt**). Previously, the context (C) stored a stack of labels, and a (**br** i) instruction would refer to the label i deep in the stack. Mirroring the changes to the semantics which store a stack of continuation tables induced by **prompt**, we now store in the context a stack of stacks of labels, as shown in the top of Figure 9 (label$((t^*)^*)^*$). Implicitly, we define the notation of context label extension C, label(t^*) used in previous WebAssembly type checking rules to mean that the label (t^*) is pushed onto the *topmost* 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 *topmost* stack in C. These implicit re-definitions allow all the other WebAssembly type checking rules 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.

The typing rules of all four instructions are shown in Figure 9.

Soundness The type checking of WebAssembly offers *soundness* as a result. Evaluation of a well-typed Wasm program is guaranteed not to get stuck with undefined redexes other than a trap (Progress), and evaluation always produces results of the expected type based on type checking (Preservation) [TODO: CITE Wasm paper] . These properties have been formally proven in [TODO: CITE HOL formalization] .

The Progress property extends easily to $\rightsquigarrow$. Formally,

Theorem 3.1 ($\rightsquigarrow$ Progress). *If* $\vdash_i s; v^*; e^* : t^*$, *then either* $e^* = v'^*$ *or* $e^* = \textbf{\textit{trap}}$ *or* $s; v^*; e^* \rightsquigarrow_i s'; v'^*; e'^*$.

The intuition for the proof is that we only have four new instructions to consider, and if these instructions have arguments of the correct types provided, then it is impossible for them to get stuck, as they will either **trap** or succeed. The typing rules added in Figure 9 guarantee that arguments of the correct types are provided. For existing WebAssembly instructions, we can invoke the prior Progress theorem. A proof is given in Theorem A.5 in the appendix.

The Preservation property is not so simple though. Naively, a direct extension of Preservation to the $\rightsquigarrow$ does not hold, because both (**control** h) and **restore** change the current stack to an unrelated stack, and the type of the new stack is unrelated to the type of the previous stack. In other words, (**control** h) and **restore** do not preserve the type of the active stack, and nor should we expect them to. However, hope is not lost as (**control** h) and **restore** do indeed preserve the types of the stacks they operated on, but these stacks are now saved in the continuation table rather than active. Thus, the Preservation property now requires relating the type of the active stack before a reduction step to the type of the stack which is now saved in the table after the reduction step. The details of the theorem statements and proofs are given in Theorem A.3 and Theorem A.4 in the appendix.

4 Achieving Safety in Practice (is this an ok name?)

4.1 Safety By Respecting Host Stack Frames [NEED FEEDBACK]

Safety of the Host Language A WebAssembly runtime environment is typically embedded in a host language, and offers an API for users of the host language to load, validate, and execute WebAssembly code. For example, Wasmtime, a JIT WebAssembly runtime implemented in Rust, offers a Rust API for calling WebAssembly functions. Since the host language (Rust in this example), requires certain semantic invariants to hold when performing foreign function calls into another language such as WebAssembly, we have to ensure that first-class continuations in WebAssembly uphold these invariants and thus cannot break the safety guarantees of the host language.

Suppose that a Rust function `foo` loads and invokes the WebAssembly function `$main`. In this case, the *machine stack* (not the formal Wasm stack) includes a stack frame for `$main` as well as a stack frame for `foo`, since control must return to `foo` when `$main` returns [2]. We call this stack a *root stack*. Suppose next that `$main` performs a (**control** h) instruction which captures the root stack (κ_R), and the handler h attempts to perform a **continuation_copy** on the root stack (κ_R). If this were allowed, then additional WebAssembly code could potentially cause the remaining code in the Rust function `foo` to be executed multiple times, which could cause for example double-free bugs and break the memory

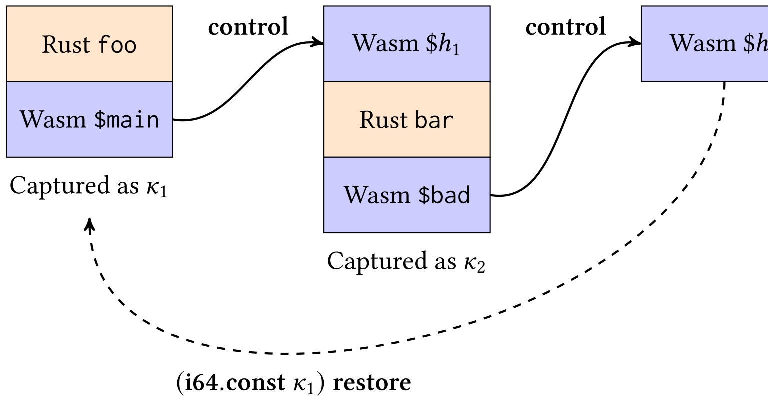

Figure 10. Restoring a continuation from a previous root stack.

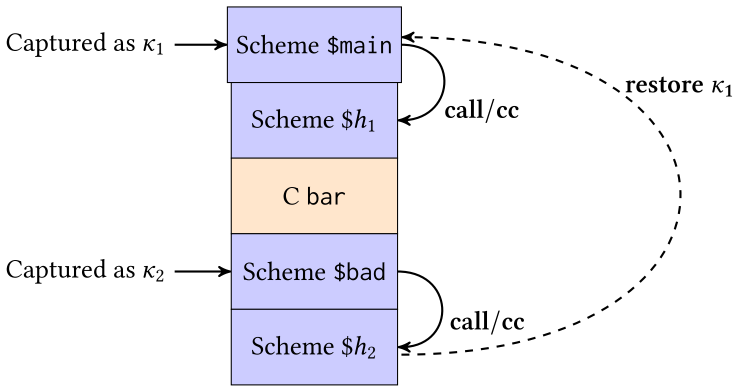

Figure 11. Restoring a continuation from a previous root stack, with call/cc.

safety of Rust. The semantics presented in §3.2 disallow a **continuation_copy** on the root stack. We now explore general principles to ensure unsafe behaviors such as this do not occur.

A general principle to obey is that when a WebAssembly function is invoked by a host function, the WebAssembly function should act like a normal function to the host function: the WebAssembly function should either return once, diverge, or **trap**. It should be impossible for the WebAssembly function to return multiple times, or for execution to complete in a separate stack and never return. A key ingredient is disallowing the copying or destruction (either via **continuation_delete** or **restore**) of a root stack, which is why the new runtime structure and rules of Figure 8 keep track of and validate at runtime the root continuation ID.

A simplified host API model WebAssembly does not include in its formal semantics any detailed semantics of the runtime environment, so it is beyond the scope of this paper

[2] if `$main` traps then an error is surfaced to `foo`

to prove that first-class continuations do not in general sacrifice the safety of the host environment. However, we can provide a simplified host API model in which we can give proofs that speak to the desired safety properties. Consider a model in which the host code invokes a single WebAssembly function once, and after that the WebAssembly state is destroyed. No host functions are exposed as callable foreign functions to the WebAssembly code. In this simplified setting, it is impossible for the called WebAssembly function to return multiple times, since once it returns the first time, WebAssembly code will never run again. A more subtle fact is to prove that if WebAssembly execution completes, then it completes in the root stack, and thus will actually return properly to the host function. The proof can be found in Theorem A.6 in the appendix.

Full host APIs In practice, a host API allows users to call WebAssembly functions multiple times, and maintain state between calls to WebAssembly. In addition, host APIs allow functions written in the host language to be exposed as callable foreign functions in WebAssembly, so that WebAssembly code can e.g. perform side effects. In order to explore the safety landscape of first-class continuations in this setting, consider the example shown in Figure 10. Execution begins with a Rust function `foo` calling a WebAssembly function `$main`. Then, `$main` does a (**control** $\$h_1$), and the `foo` root stack is captured in continuation κ_1. At this point, the handler $\$h_1$ is now executing in a new stack. After that, $\$h_1$ calls a Rust function `bar`, which then calls another WebAssembly function `$bad`.

Now, since `$bad` was called from the Rust `bar`, we need to ensure that `bar` will be returned to. If `$bad` is allowed to execute (**i64.const** κ_1) **restore**, then the second stack which includes the stack frame of the Rust function `bar` would be deleted, which is unsafe. Thus, we might like to treat both the first stack and this new second stack as root stacks, so that executing (**i64.const** κ_1) **restore** inside `$bad` is disallowed. However, this restriction is not sufficient. Suppose that `$bad` performs a (**control** $\$h_2$) instruction as shown in Figure 10 (which is safe), and then the handler $\$h_2$, which is running in a fresh non-root stack, executes (**i64.const** κ_1) **restore**. Since $\$h_2$ is running in a non-root stack, it is allowed to execute the (**i64.const** κ_1) **restore** instruction. While this does not delete the stack frame of `bar`, it does mean that `$main` and then `foo` will continue execution, without `bar` ever having been returned to. Since `bar` is not returned to, but execution continues, this can easily lead to a memory leak in Rust, and is unsafe.

Solution The spot in which the above example went awry is in the **restore**κ_1 instruction. The key property which is violated is that first-class continuation instructions should only be able to touch continuations which have been created since the most recent host function call. Concretely, the **restore** in handler $\$h_2$ above should be disallowed from touching

```
void f() {
    int x = 0;
    int *xp = &x;
    fork(); // similar to uniform() in Figure 5
    x++;
    printf("%d\n", x);
}
```

Figure 12. C code which writes to a stack address

κ_1, since κ_1 was created by the first root stack, before the call the Rust function `bar`. The same restriction should hold for **continuation_copy** and **continuation_delete**. In order to enforce this, we adopt a stacked model of first-class continuations: a stack of continuation tables is maintained, with a fresh continuation table pushed onto the stack for each host function call. As before, we now have a single unique root continuation, now *per continuation table*. When the root continuation returns (as guaranteed by Theorem A.6), the continuation table is popped from the stack. During execution, a **restore** (or **continuation_copy** or **continuation_delete**) instruction can only index into the continuation table at the top of the stack.

Since instructions can only index into the continuation table on top of the table stack, and a fresh continuation table is pushed onto the stack when a host function invokes a WebAssembly function, there is no shared continuation state accessible among re-invocations of WebAssembly from host functions. This stacked model of continuation tables transforms the complex case of host APIs allowing shared state between function invocations and host functions being callable from WebAssembly to the simplified host API model discussed above, which we know to be safe. Formalizing this stacked model is beyond the scope of this work, since doing so requires the semantics of WebAssembly to know when a host function vs. a non-host function is called, which is a significant change to the semantics of WebAssembly. However, the semantics presented in Figure 8 are designed to be easily compatible with this model: the administrative functions cont(s, i, k) and root(s, i) simply need to index only into the continuation table at the top of the table stack.

4.2 Accommodating Shadow Stacks [NEED FEEDBACK]

[Note: shadow stacks don't need to be introduced here, as they should be introduced in the intro. – Donald] [Shadow stack isn't really the right vocab, not sure what to use instead though. – Donald]

Current compilers for most practical programming languages to WebAssembly are forced to make use of a shadow stack in order to implement certain features of the language. For example, the Go to WebAssembly compiler uses shadow stacks in order to implement garbage collection and green threads, while the Emscripten C/C++ to WebAssembly compiler uses a shadow stack to compile C code which takes pointers to values on the stack.

When the (**control** h) WebAssembly instruction is used to capture the current continuation, the shadow stack is not captured, since (**control** h) has no way to know about the shadow stack which is stored arbitrarily in linear memory. Therefore, it is not correct to simply compile `control(h)` in C directly to (**control** h) in WebAssembly.

For example, consider the code shown in Figure 12. Line 4 uses mechanisms almost identical to `uniform` in Figure 6 to create two copies of the continuation in the middle of `f`, and execute both. Writing to the variable `x` should only change the currently executing continuation's copy. However, because of line 3, the Emscripten compiler places `x` on the shadow stack in linear memory rather than on the WebAssembly stack, and thus a copy of `x` is not made, if using the above incorrect compilation strategy.

To remedy this, when compiling `control(h)` we insert some additional code which captures the C shadow stack. [Not sure how much to say about this... – Donald].

5 Efficient Implementation in a JIT Runtime

[[[TODO: Change this:]]] To validate the efficiency and practicality of our proposed first-class continuation extension to Wasm, we implement it by extending Wasmtime, a real-world JIT runtime for Wasm. [TODO: CITE] Wasmtime [[[TODO: Explain more about what Wasmtime is used for, why its interesting / practical]]] . Below we describe our experience implementing the proposed extension in Wasmtime. In §6 we evaluate our implementation by exploring its performance and expressivity on various examples.

A JIT for Wasm, such as Wasmtime, must allocate registers of the target machine (e.g. x86-64) for various values of the stack, and compile the structured control flow constructs of Wasm to assembly-level labels and branches. Thus, at runtime we no longer have access to the Wasm stack as specified in the formal semantics, as it is now split into register values, labels and branch instructions, and a traditional *machine stack* used for spilling local variables and performing function calls. Therefore, saving the entire formal Wasm stack corresponds at runtime in a JIT implementation to saving the registers, the instruction pointer, and the machine stack. Since the JIT compiles Wasm code to arbitrary assembly code, the implementation for saving and restoring the Wasm stack depends on the target platform, as well as on the specific code that the JIT generates for other instructions. In particular, the calling convention that the JIT follows affects which registers need to be saved and restored.

To implement the proposed extension, we compile (**control** j), **restore**, **continuation_copy** and **continuation_delete** to function calls to the `_control`, `_restore`, `_continuation_copy` and `_continuation_delete` runtime functions (with appropriate arguments). In order to be able to capture and restore the continuation, `_control` and `_restore` are written in platform-dependent assembly (x86-64 is supported currently), while `_continuation_copy` and `_continuation_delete` can be written in C, though their implementation is still dependent on the specific implementation details of the JIT. It is then up to the code of those functions to properly manage the registers, instruction pointer, and machine stack in order to achieve the correct first-class continuation semantics.

***Implementation of* `_control`** The implementation of `_control` takes as arguments a function pointer to the address of the emitted code of immediate function argument j in the (**control** j) Wasm instruction, and an arbitrary i64 argument v. First, `_control` finds a free entry in its table of continuations via a free list. The index into the table will be κ, the continuation ID. Next, `_control` must save appropriate registers into the table. The registers that need to be saved are dependent on the calling convention implemented in the JIT. A register r which is a caller-saved register need not be saved inside `_control`, as the JIT would have already emitted code that saves r into the machine stack before the call to `_control`. At a minimum, the stack pointer and the instruction pointer both need to be saved. After saving registers, `_control` is now ready to invoke the control handler j given as an immediate argument. To do so, `_control` first allocates a new, separate stack, switches the stack pointer to use the new stack, and then jumps to the function pointer for j, passing κ and v as arguments to the function. The allocation and switching to a new stack is fast, since we preallocate plenty of empty stack spaces, and switch to one on demand, causing the stack to be lazily paged in to memory.

***Implementation of* `_restore`** The implementation of `_restore` works in reverse to `_control`. Given continuation ID κ and arbitrary argument v as input, `_restore` first indexes into the table of continuations with index κ. At this point, `_restore` must perform a dynamic check to ensure that the entry in the table is non-null, so as to enforce the one-shot continuation semantics. If this passes, then all registers other than the instruction pointer are restored. In particular, the stack pointer is restored. Next, `_restore` marks both the continuation table entry and the previous stack as free, and jumps to the previously saved instruction pointer, with the value v as a return value.

***Implementation of* `continuation_copy/delete`** The implementation of `continuation_copy` is conceptually straightforward, but has some subtleties depending on the implementation of the JIT. The basic implementation of `continuation_copy` works by taking as input the ID κ, allocating a new ID κ', copying the context saved at index κ into index κ', and then returning κ'. To copy the context over, we have to decide how to copy the saved register values over, and how to copy the stack over. The semantics require allocating a new stack associated with κ', so that when the continuation κ' is later restored to, it will not modify the stack of κ. After allocating this new stack, we need to copy the old stack contents over into the new stack contents. Finally, for the registers we copy their values over directly into index κ', except that the stack pointer must be updated to point at the correct position inside the new stack. The implementation of `continuation_delete` is the simplest of all four f unctions: we just have to mark the continuation ID and stack as free. This is our implementation, and for Wasmtime it works correctly.

However, depending on the specifics of the JIT, this implementation of `_continuation_copy` may not be correct. This can generally go wrong if either the registers or the stack contents contain memory references to somewhere in the old stack. If this is the case, then the copied continuation κ' may erroneously perform loads or stores on the stack for κ. Unlike C, Wasm does not allow the user to obtain memory references to values in the stack, so typically a Wasm JIT would not emit code which puts memory references to the stack in registers or on the stack. However, one common specific case of this is if the JIT emits the use of the base pointer register, which stores the address of the top of the stack frame, and is pushed onto the stack, forming a base pointer chain throughout the stack. This happens frequently in real systems, as for example Clang/LLVM uses the base pointer register when compiling C code. If the WebAssembly JIT were to do this, both the base pointer register, and all of the base pointers pushed onto the stack would incorrectly reference the old stack for κ. In order to fix this, the base pointer register would have to be updated to point to the correct relative address in the new stack, and then the base pointer chain must be walked through the new stack, and updated to point to the correct new addresses. Fortunately, Wasmtime does not emit use of the base pointer, so this is not needed in our implementation.

***Implementation of* `prompt`** TODO: pretty straightforward

6 Evaluation [NEED REWRITE]

Our goal of `Wasm/k` is to extend WebAssembly so that languages such as Go can compile runtime features such as user-level threads to efficient code. We have two considerations of performance to explore.

First, current solutions for compiling threads generate complex WebAssembly code which introduces loads and stores for saving variables in a shadow stack in user memory, and additional branches in order to simulate a state machine for threading. These loads/stores and branches are on all code paths, and thus add a performance overhead even for code which makes light or no use of threads. §2.1 demonstrates a short case study which highlights these code generation issues. `Wasm/k` eliminates or reduces the need for these complex code generation techniques.

Second, there is the performance of the instructions introduced by `Wasm/k`. §6.2 explores the performance of `Wasm/k` in the context of implementing threads, in comparison to Asyncify.

6.1 Performance Characteristics of `Wasm/k`

[With the suckiness of Go->Wasm shown, this section now aims to show that `Wasm/k` can fix the specific causes of slowness from above – Donald]

Notes:

1. Do threading benchmarks using C->Wasm (threading disabled), C->Wasm (kthreads), C->Wasm (asyncify threads), equivalent Go, all using standard infrequent yields.
2. Looking at performance, threading disabled and kthreads should be close, asyncify and Go slower.
3. Look at perf counters, again (threading disabled and kthreads) similar, (asyncify and Go) similar

6.2 Performance Evaluation

As discussed in Section [[[TODO: insert section here]]] existing implementations of non-local control operators in Wasm are implemented by a source-to-source rewriting strategy. The primary existing implementation is Asyncify [TODO: CITE https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html] which adds four pseudo-operators to Wasm that allow for unwinding and rewinding the stack. In a compilation pass, Asyncify transforms Wasm code that uses these pseudo-operators to standard Wasm.

In this microbenchmark we compare the performance of a cooperative user-level threading library built using Asyncify and using `Wasm/k`. Both implementations use the exact same thread scheduler (simple round-robin) and are written directly in Wasm, to ensure that they are as identical as possible. The only significant difference is in the implementation of the `$yield` function, which is implemented either using Asyncify or `Wasm/k` primitives. All experiments are run on a 2 GHz Quad-Core Intel Core i7 with 64 GB RAM and Hyper-Threading enabled.

Both implementations are tested on the exact same computational task: approximating π by summing 2^{28} terms from an infinite series, distributed equally among 16 threads. Both implementations use identical Wasm code for the computation of these terms. As it is a cooperative threading library, we can choose when to trigger a thread yield. We trigger a thread yield every $I = 2^i$ terms computed, where i is fixed during runtime but varies across multiple experiment trials. For each i, 10 trials are conducted, and total running time is measured.

The results are shown in ??. By varying the frequency of thread yields, we can measure two aspects of the performance of the implementations: the performance of conducting the yield operation itself, and the performance impact on the rest of the code caused by the implementation. As the frequency of yields decreases, the performance of both implementations asymptotes to the performance of the non-yield parts of the code. In this regime, our implementation is $\approx$ 1.3x faster than Asyncify. This shows that the source-to-source rewriting strategy of Asyncify forces programmers to pay a large performance cost for code which rarely use stack manipulation primitives. [[[TODO: Should I show Go Wasm experiments that show that Go has to pay the same cost?]]] `Wasm/k` has no such cost for non-yields parts of the code, as the implementation of `Wasm/k` does not affect the code generation of other Wasm instructions. With the most frequent yields, once every term, our implementation is $\approx$ 1.6x faster than the Asyncify implementation. Thus, our implementation is also faster at performing the actual thread context switching.

7 Related Work [NEED TO WRITE]

8 Conclusion [NEED TO WRITE]

A Technical Appendix

Lemma A.1 ($\hookrightarrow$ Preservation). *If* $\vdash_i s; v^*; e^* : t^*$ *and* $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$*, then* $\vdash_i s'; v'^*; e'^* : t^*$

Proof. Suppose that $\vdash_i s; v^*; e^* : t^*$ and $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$. The reduction $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$ has five redex cases to consider:

Case 1, Rule (3a): The reduction must be:

$$s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_copy} \hookrightarrow_i s'; v^*; (\textbf{i64.const } \kappa')$$

Then, $\vdash_i s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_copy} : \textbf{i64}$ and $\vdash_i s'; v^*; (\textbf{i64.const } \kappa') : \textbf{i64}$.

Case 2, Rule (3b): The reduction must be:

$$s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_copy} \hookrightarrow_i s; v^*; \textbf{trap}$$

Then, $\vdash_i s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_copy} : \textbf{i64}$ and $\vdash_i s'; v^*; \textbf{trap} : \textbf{i64}$.

Case 3, Rule (4a): The reduction must be:

$$s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_delete} \hookrightarrow_i s; v^*; \epsilon$$

Then, $\vdash_i s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_delete} : \epsilon$ and $\vdash_i s'; v^*; \epsilon : \epsilon$.

Case 4, Rule (4b): The reduction must be:

$$s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_delete} \hookrightarrow_i s; v^*; \textbf{trap}$$

Then, $\vdash_i s; v^*; (\textbf{i64.const } \kappa)\ \textbf{continuation\_delete} : \epsilon$ and $\vdash_i s'; v^*; \textbf{trap} : \epsilon$.

Case 5: Otherwise, the redex is an original Wasm instruction. In this case, the original Wasm preservation theorem holds, giving that $\vdash_i s'; v'^*; e'^* : t^*$.

□

Definition A.2. $\vdash_i s; \text{cont}(s, i, \kappa) : t^*$ is defined as a shorthand for type checking with a dummy value substituted into the context: $\vdash_i s; c.\text{locals}; c.\text{ctx}[(\textbf{i64.const } 0)] : t^*$ where $c = \text{cont}(s, i, \kappa)$.

Theorem A.3 ($\rightsquigarrow$ Root Preservation). *If* $\vdash_i s; v^*; e^* : t^*$ *and* $\text{root}(s, i) = \textit{nil}$ *and* $s; v^*; e^* \rightsquigarrow_i s'; v'^*; e'^*$*, then either:*

G1) $\vdash_i s'; v'^*; e'^* : t^* \wedge \text{root}(s', i) = \textit{nil}$ *or*
G2) $\vdash_i s'; v'^*; e'^* : t'^* \wedge \text{root}(s', i) = \kappa_R \wedge \vdash_i s'; \text{cont}(s', i, \kappa_R) : t^*$

Proof. Suppose the following hypotheses hold:

H1) $\vdash_i s; v^*; e^* : t^*$
H2) $\text{root}(s, i) = \textit{nil}$
H3) $s; v^*; e^* \rightsquigarrow_i s'; v'^*; e'^*$

By H3 there are five cases to consider:

Case 1, Rule (1a): The reduction must be:

$$s; v^*; L^{\max}[(\textbf{i64.const } v)(\textbf{control } h)] \rightsquigarrow_i s'; \epsilon; (\textbf{i64.const } \kappa)(\textbf{i64.const } v)(\textbf{call } h)\textbf{trap}$$

By H2 and Rule (1a), $\text{root}(s', i) = \kappa$. By the typing rule of $(\textbf{control } h)$, h has function type $\textbf{i64 i64} \to \epsilon$, so $\vdash_i s'; \epsilon; (\textbf{i64.const } \kappa)(\textbf{i64.const } v)(\textbf{cal}$
t'^* for some t'^*. Finally, since $\vdash_i s; v^*; L^{\max}[(\textbf{i64.const } v)(\textbf{control } h)] : t^*$ by H1, then $\vdash_i s; v^*; L^{\max}[(\textbf{i64.const } 0)] : t^*$, by inspecting the typing rule of $(\textbf{control } h)$. Therefore, goal G2 is satisfied.

Case 2, Rule (1b): The reduction must be:

$$s; v^*; L^{\max}[(\textbf{i64.const } v)(\textbf{control } h)] \rightsquigarrow_i s; v^*; \textbf{trap}$$

We can trivially conclude that $s; v^*; \textbf{trap} : t^*$, and $\text{root}(s, i) = \textit{nil}$. Case complete by goal G1.

Case 3, Rule (2a): This case is impossible since $\text{root}(s, i) = \textit{nil}$ by H2.

Case 4, Rule (2b): The reduction must be:

$$s; v^*; L^{\max}[(\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)\textbf{restore}] \rightsquigarrow_i s; v^*; \textbf{trap}$$

We can trivially conclude that $s; v^*; \textbf{trap} : t^*$, and $\text{root}(s, i) = nil$. Case complete by goal G1.

Case 5, Rule [No-Ctrl]: There exists a reduction:

$$s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$$

By an inspection of the semantic rules for $\hookrightarrow_i$, it is clear that $\text{root}(s', i) = \text{root}(s, i) = nil$. By Lemma A.1 and H1, $\vdash_i s'; v'^*; e'^* : t^*$. Case complete by goal G1.

□

Theorem A.4 ($\rightsquigarrow$ Non-Root Preservation). *If* $\vdash_i s; v^*; e^* : t^*$ *and* $\text{root}(s, i) = \kappa_R$ *and* $s; v^*; e^* \rightsquigarrow_i s'; v'^*; e'^*$ *and* $\vdash_i s; \text{cont}(s, i, \kappa_R) : t^*_R$ *and all saved continuations* κ *are well-typed, then either:*

G1) $\vdash_i s'; v'^*; e'^* : t'^* \wedge \text{root}(s', i) = \kappa_R$ *or*
G2) $\vdash_i s'; v'^*; e'^* : t^*_R \wedge \text{root}(s', i) = nil$

Proof. Suppose the following hypotheses hold:

H1) $\vdash_i s; v^*; e^* : t^*$
H2) $\text{root}(s, i) = \kappa_R \neq nil$
H3) $s; v^*; e^* \rightsquigarrow_i s'; v'^*; e'^*$
H4) $\vdash_i s; \text{cont}(s, i, \kappa_R) : t^*_R$
H5) $\forall \kappa. \text{cont}(s, i, \kappa) \neq nil \implies \exists t^*_\kappa.\ \vdash_i s; \text{cont}(s, i, \kappa) : t^*_\kappa$

By H3 there are five cases to consider:

Case 1, Rule (1a): The reduction must be:

$$s; v^*; L^{\max}[(\textbf{i64.const}\ v)(\textbf{control}\ h)] \rightsquigarrow_i s'; \epsilon; (\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)(\textbf{call}\ h)\textbf{trap}$$

By H2 and Rule (1a), $\text{root}(s', i) = \kappa_R$. By the typing rule of $(\textbf{control}\ h)$, h has function type $\textbf{i64 i64} \rightarrow \epsilon$, so $\vdash_i s'; \epsilon; (\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)($
t'^* for some t'^*. Therefore, goal G1 is satisfied.

Case 2, Rule (1b): The reduction must be:

$$s; v^*; L^{\max}[(\textbf{i64.const}\ v)(\textbf{control}\ h)] \rightsquigarrow_i s; v^*; \textbf{trap}$$

We can trivially conclude that $s; v^*; \textbf{trap} : t^*$, and $\text{root}(s, i) = \kappa_R$. Case complete by goal G1.

Case 3, Rule (2a): The reduction must be:

$$s; v^*; L^{\max}[(\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)\textbf{restore}] \rightsquigarrow_i s'; v'^*; L^{\max\prime}[(\textbf{i64.const}\ v)]$$

where $L^{\max\prime} = \text{cont}(s, i, \kappa).\text{ctx}$ and $v'^* = \text{cont}(s, i, \kappa).\text{locals}$. There are two sub-cases to consider:

Sub-case 1, $\kappa = \kappa_R$ Suppose that $\kappa = \kappa_R$. Then by Rule (2a) $\text{root}(s', i) = nil$. By H4, $\vdash_i s'; v'^*; L^{\max\prime}[(\textbf{i64.const}\ v)] : t^*_R$. Sub-case complete by goal G2.

Sub-case 2, $\kappa \neq \kappa_R$: Suppose that $\kappa \neq \kappa_R$. Then by Rule (2a) $\text{root}(s', i) = \kappa_R$, and $\text{cont}(s, i, \kappa) \neq nil$. By H5, $\exists t^*_\kappa$ such that $\vdash_i s; \text{cont}(s, i, \kappa) : t^*_\kappa$. Therefore, $\vdash_i s'; v'^*; L^{\max\prime}[(\textbf{i64.const}\ v)] : t^*_\kappa$. Sub-case complete by goal G1.

Case 4, Rule (2b): The reduction must be:

$$s; v^*; L^{\max}[(\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)\textbf{restore}] \rightsquigarrow_i s; v^*; \textbf{trap}$$

We can trivially conclude that $s; v^*; \textbf{trap} : t^*$, and $\text{root}(s, i) = \kappa_R$. Case complete by goal G1.

Case 5, Rule [No-Ctrl]: There exists a reduction:

$$s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$$

By an inspection of the semantic rules for $\hookrightarrow_i$, it is clear that $\text{root}(s', i) = \text{root}(s, i) = \kappa_R$. By Lemma A.1 and H1, $\vdash_i s'; v'^*; e'^* : t^*$. Case complete by goal G1.

□

Theorem A.5 ($\rightsquigarrow$ Progress). *If* $\vdash_i s; v^*; e^* : t^*$*, then either* $e^* = v'^*$ *or* $e^* = \textbf{trap}$ *or* $s; v^*; e^* \rightsquigarrow_i s'; v'^*; e'^*$.

Proof. Suppose that $\vdash_i s; v^*; e^* : t^*$. Figure 9 adds four new cases for the typing, in addition to the original cases from the Wasm typing relation. It suffices to prove the goal for redexes corresponding to each typing rule, as otherwise a single-step reduction will trivially occur previously in the stack via the inductive hypothesis.

Case 1, (**control** h): The stack e^* is of the form $e^* = L^{\max}[(\textbf{i64.const}\ v)(\textbf{control}\ h)]$, with $C_{\text{func}}(h) = \textbf{i64 i64} \longrightarrow \epsilon$. The store s has a continuation table $\text{cont}(s, i, \cdot)$. There are two possibilities: either there exists an ID κ such that $\text{cont}(s, i, \kappa) = \mathit{nil}$, or there exists no such κ. If such a κ exists, then by Rule (1a):

$$s;\ v^*;\ L^{\max}[(\textbf{i64.const}\ v)\ (\textbf{control}\ h)] \rightsquigarrow_i s';\ \epsilon;\ (\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ (\textbf{call}\ h)\ \textbf{trap}$$

with s' given as in Rule (1a). If there is no such κ, then by Rule (1b):

$$s;\ v^*;\ L^{\max}[(\textbf{i64.const}\ v)\ (\textbf{control}\ h)] \rightsquigarrow_i s;\ v^*;\ \textbf{trap}$$

Either way, the case is complete.

Case 2, **restore**: The stack e^* is of the form $e^* = L^{\max}[(\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)\textbf{restore}]$. There are two possibilities for the stored continuation. If $\text{cont}(s, i, \kappa) = \mathit{nil}$, then by Rule (2b):

$$s; v^*; L^{\max}[(\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)\textbf{restore}] \rightsquigarrow_i s; v^*; \textbf{trap}$$

If $\text{cont}(s, i, \kappa) = \{\text{locals } = v^{*\prime}, \text{ctx } = L^{\max\prime}\}$, then by Rule (2a):

$$s; v^*; L^{\max}[(\textbf{i64.const}\ \kappa)(\textbf{i64.const}\ v)\textbf{restore}] \rightsquigarrow_i s'; v^{*\prime}; L^{\max\prime}[(\textbf{i64.const}\ v)]$$

with s' given as in Rule (2a).

Case 3, **continuation_copy**: The stack e^* is of the form $e^* = Ł^k[(\textbf{i64.const}\ \kappa)\textbf{continuation\_copy}]$. Based on the state of s and index κ, by Rules (3a) and (3b) the stack will either step to $Ł^k[(\textbf{i64.const}\ \kappa')]$ for some κ', or **trap**.

Case 4, **continuation_delete**: The stack e^* is of the form $e^* = Ł^k[(\textbf{i64.const}\ \kappa)\textbf{continuation\_delete}]$. Based on the state of s and index κ, by Rules (4a) and (4b) the stack will either step to $Ł^k[\epsilon]$, or **trap**.

□

Theorem A.6 (Root Continuation Completion). *If* $\vdash_i s; v_l^*; e^* : t^*$ *and* $\text{root}(s, i) = \mathit{nil}$ *and* $s; v_l^*; e^* \rightsquigarrow_i^* s'; v_l'^*; v^*$*, then* $\text{root}(s', i) = \mathit{nil}$.

Proof. The multi-step reduction $s; v_l^*; e^* \rightsquigarrow_i^* s'; v_l'^*; v^*$ can be viewed as some number of single-step reductions caused by (**control** h), **restore**, **continuation_copy**, or **continuation_delete**, interleaved with other non-control instruction reductions. Since the reduction $s; v_l^*; e^* \rightsquigarrow_i^* s'; v_l'^*; v^*$ has finite length, and the only instructions which increase the total number of saved continuations in the continuation table are (**control** h) and **continuation_copy**, there exist only finitely many continuations in the continuations table at the end of the multi-step reduction. Thus, $\text{root}(s', i)$ must be either nil or κ_R. If $\text{root}(s', i)$ is nil, then the proof is complete. Otherwise, suppose $\text{root}(s', i) = \kappa_R$. Then, the current stack must be a non-root stack. Since non-root stacks can only be created via (**control** h) or **continuation_copy**, any non-root stack is terminated by a **trap** instruction, which is a contradiction. □