Efficient Compilation of Tail Calls and Continuations to JavaScript

Eric Thivierge Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

This paper describes an approach for compiling Scheme's tail calls and first-class continuations to other dynamic languages without those features, with a focus on JavaScript. Our approach is based on the use of a custom virtual machine intermediate representation which is translated to the target language. We compare this approach, which is used by the Gambit-JS compiler, to the Replay-C algorithm, used by Scheme2JS (a derivative of Bigloo), and CPS conversion, used by Spock (a derivative of Chicken). We analyse the performance of the three approaches with a set of benchmark programs. Our approach is consistently faster than the others on the benchmark programs, and up to two orders of magnitude faster when first-class continuations are used intensively.

Keywords Continuation, tail call, Scheme, JavaScript, trampoline, virtual machine

1. Introduction

There is an increasing trend in implementing programming languages as compilers to other high-level programming languages. Such a compiler gives increased portability, allowing the source language to execute wherever the target language can be executed, it gives more direct access to the features available in the target language (libraries, tools, etc), and it makes it easier to integrate program parts written in the source language with an existing code base in the target language. Some of the more popular target languages are C, C++ and Java, or more specifically the JVM bytecode. Recently, JavaScript has also become a popular target due to the unique role it plays in web browsers and web applications. Currently, there are over 50 compilers [2] targetting JavaScript, including compilers whose source language is C, C++, Java, Python, Haskell, and Scheme.

Using a popular dynamic language such as JavaScript, Python and Ruby as the target language for a Scheme compiler is alluring because they offer dynamic typing, introspective features, closures and garbage collection which simplify the translation of Scheme which also has those features. The biggest challenge remaining is the implementation of tail calls and first-class continuations which have no direct equivalent in the general case in these target languages. The support for first-class continuations, and in particular serializable continuations, is a useful feature for implementing continuation based web frameworks, distributed programming languages supporting process migration (e.g. Termite [9]), and threaded applications. By targetting JavaScript, such applications could also execute in web browsers.

A Scheme system which conforms to the standard must implement tail calls without stack growth, and it must implement the `call/cc` predefined procedure which captures implicit continuations so that they can be invoked explicitly, possibly multiple times. Because of the complexity and run-time cost of implementing these features in a high-level target language, some systems reduce their generality by default (for example only transforming self tail calls using loops, and one-shot escape continuations using exceptions), and support the full generality only through special compilation options.

Various approaches for implementing these features in their full generality have been used in Scheme compilers targetting high-level languages such as the Bigloo [14], Chicken [1] and Gambit-C [7] Scheme to C compilers, and the Scheme2JS [13] [12] [11], Spock [3], and Whalesong [15] Scheme to JavaScript compilers.

Tail calls can be implemented with trampolines to avoid stack growth when one function jumps to another function. Trampolines are used in Scheme2JS and Gambit-C.

The approach used by Chicken, Spock, and Whalesong, known as Cheney on the MTA [4], implements tail calls with normal calls and uses a non-local escape mechanism, such as `throw/catch` or C's `setjmp/longjmp`, at appropriate moments to reclaim the useless stack frames in bulk (recent versions of Spock unwind the stack using a cascade of simple function returns to avoid compatibility issues with some browsers). The Cheney on the MTA approach is normally combined with a CPS conversion of the code so that all stack frames can be reclaimed. Indeed all calls are tail calls in the CPS'ed code. As an interesting side-effect, the implementation of first-class continuations is greatly simplified since all translated functions receive an explicit continuation function.

In Scheme2JS, first-class continuations are implemented by copying the stack frames to the heap. In a high-level language like JavaScript where the stack can't be accessed directly, exceptions can be used to visit the stack frames iteratively (from newest to oldest) to build a copy in the heap and reclaim the stack frames. The code generated for functions is structured in such a way that the original stack frames are recreated by a traversal of the copy in the heap (in other words functions contain code to save their frames and also recreate them, depending on whether a continuation is being captured or invoked). This is known as the Replay-C algorithm in [12].

Gambit-C, which uses a virtual machine based intermediate language, models the stack explicitly, as a C array, and consequently could implement first-class continuations with most of the algo-

rithms used by native code compilers [5]. A fine grained variant of the Hieb-Dybvig-Bruggeman strategy [10] is used.

These approaches offer different tradeoffs. It is convenient to consider as a baseline a direct translation of Scheme to the target language which ignores tail calls and first-class continuations, and to look at the overhead over the baseline which results from a particular approach. Being based on a CPS conversion, the Cheney on the MTA approach makes first-class continuation capture and invocation very fast, but it slows down non-tail calls due to the overhead of creating the closure for the continuation, passing it to the called function, and the added pressure on the garbage collector. Scheme2JS is designed to favor programs with infrequent first-class continuation operations. The Replay-C algorithm which it uses has more work to do when first-class continuations are captured and invoked, for copying between the stack and heap. Programs which seldom capture and invoke continuations still have an overhead for the `try/catch` forms which wrap all the non-tail calls, but it is possibly less work than creating continuation closures and garbage collecting them, which of course depends on the technology used to implement the target language, which has evolved since Scheme2JS's creation. Modeling the stack explicitly as in Gambit-C also has an overhead because accesses to source language variables are converted, in the general case, to target language array indexing operations of the stack. All the approaches generate code with a more complex structure than the baseline, which also causes overhead because optimization by the target VM is hindered.

In this paper we will demonstrate that an approach for implementing tail calls and first-class continuations which is based on a virtual machine strikes a good balance between simplicity of implementation and performance. The JavaScript back-end we have developed for Gambit generates code which is faster than both Spock and Scheme2JS, sometimes by over two orders of magnitude when continuations are used extensively.

For simplicity, we will explain our approach using JavaScript as the target language. The back-ends we have developed for Python and Ruby are essentially the same, save for surface syntax, and we will mention the main differences in Section 6.

2. Gambit Virtual Machine

The Gambit compiler front-end follows a fairly standard organization as a pipeline of stages which parse the source code to construct an AST, expand macros, apply various program transformations on the AST (assignment conversion, lambda lifting, inlining, constant folding, etc), translate the AST to a control flow graph (CFG), and perform additional optimizations on the CFG. The front-end does not perform a CPS conversion. Finally the target language specific back-end converts the CFG to the target language.

2.1 Instruction Set

The CFG is a directed graph of basic blocks containing instructions of a custom designed virtual machine [8], called the Gambit Virtual Machine (GVM). The GVM is a simple machine with a set of locations in which any Scheme object can be stored: general purpose registers (e.g. `r2`), a stack of frames (e.g. `frame[2]` is the second slot of the topmost stack frame), and global variables. The front-end will generate GVM code which respects the back-end specific constraints, such as the number of available GVM registers, the calling convention, and the set of inlinable Scheme primitives. In the current back-ends, there are 5 GVM registers, and the calling convention passes in registers the return address (in `r0`) and the last 3 arguments (in `r1`, `r2`, and `r3`).

There are seven GVM instructions: *label*, *jump*, *ifjump*, *switch*, *copy*, *apply*, and *close*.

Each basic block begins with a *label* instruction which identifies the basic block and indicates its kind. There are local blocks and first-class blocks (plain function entry point, closure entry point, function call return point). References to first-class blocks can be stored in any GVM location.

The last instruction of a basic block is a branch instruction which transfers control to another basic block, either unconditionally (*jump*), or conditionally (*ifjump* or *switch*). Conditional and unconditional branches can branch to local blocks. Only *jump* instructions can branch to first-class blocks. In general, function calls are implemented with a *jump* instruction which indicates the argument count. The *label* instruction for the entry point indicates the function's number of parameters and whether or not there is a rest parameter, allowing at function entry a dynamic check of the argument count and the creation of the rest parameter. Function calls to known local functions without a rest parameter avoid the argument count check (they become *jump*s without argument count to local blocks). Scheme's `if` and `case` forms are respectively implemented using the *ifjump* and *switch* instructions which branch to one of multiple local blocks.

Data movement and primitive operations (e.g. `cons`) are respectively performed with the *copy* and *apply* instructions. These instructions indicate the destination GVM location, and the source operands, which can be any GVM location, immediate Scheme object or reference to a first-class block.

Finally, the *close* instruction creates a group of one or more flat closures. For each closure is specified the closure's entry point, the values of the closed variables, and the destination GVM location where the closure reference is stored. Mutually referential closures, which `letrec` can create, can be constructed because the assignment to the destinations are conceptually performed after the closures are allocated but before the content of the closure is initialized. A *jump* to a closure reference will transfer control to the closure entry point contained in the closure and automatically store the closure reference in the *self register*, which is the last GVM register, i.e. `r4`, in the current back-ends. The closed variables are accessed indirectly using the closure reference.

2.2 Stack Frame Management

The GVM does not expose a stack-pointer register, or push/pop instructions. The allocation and deallocation of stack frames is specified implicitly in the *label* and branch instructions. The *label* instruction indicates the topmost frame's size immediately after it has been executed. Similarly, the branch instruction at the end of the basic block indicates the frame size at the transfer of control. The difference between the exiting and entering frame sizes corresponds to the amount of stack space allocated (or deallocated if the difference is negative). The back-end can generate a single stack pointer adjustment at every GVM branch instruction. Moreover, the back-end can use the entering stack frame size to calculate the offset to add to the stack pointer to access a given stack slot, which are indexed from the base of the frame.

Tail and non-tail calls must pass arguments to the called function on the stack and in registers. The arguments on the stack are known as the *activation frame*. It is empty if few arguments are passed. A *continuation frame* is created for non-tail calls to store the values needed upon return from the call at the return point. The continuation frame always contains the return address of the function which created the continuation frame.

When a GVM branch instruction corresponds to a tail call, the topmost stack frame only contains the activation frame. In the case of a non-tail call, the stack frame includes both the activation frame and, below it, the continuation frame. When the branch corresponds to a function return, the stack frame is empty.

In general, a runtime system for the GVM may use a limited size memory area for allocating stack frames. This does not imply that recursion depth is limited. Indeed, when the stack area overflows a

new stack area could be allocated from the heap or the stack frames it contains could be copied to the heap. Either way it is necessary to detect these overflows and then call a stack overflow handler.

The GVM provides for this through the more general concept of *interrupt*. An interrupt is an event, such as a stack area overflow, heap overflow, and preemptive multithreading timer interrupt, which disrupts the normal sequence of execution. The GVM polls for interrupts using *interrupt checks* which are spread throughout the code. GVM branch instructions carrying a *poll* flag perform interrupt checks. Before the transfer of control, the presence of an interrupt is checked and an appropriate handler is called if an interrupt is detected. Note that combining the poll operation with the branch instruction provides some optimization opportunities: the branch destination can be the destination of the target language conditional branch in the case of an interrupt check failure.

The front-end guarantees that the frame size grows by at most one slot per GVM instruction and also that the number of GVM instructions executed between poll points is bounded by the constant L, the maximum poll latency (see [6] for details). Consequently, the bounds of the stack area will never be exceeded if an extra L slots are reserved at the end of the stack area. This is called the stack fudge area.

2.3 Example

To illustrate the operation of the front-end and specifically the management of the stack, consider the function `foreach` whose source code is given in Figure 1. This function contains both a tail call to `loop` and a non-tail call to `f`. To make the GVM code generated easier to read, declarations are used in the source code to ensure that the primitive functions `pair?`, `car`, and `cdr` get inlined, and dynamic type checks are not performed by `car` and `cdr`, and the loop is not unrolled.

```
1.  (declare (standard-bindings)
2.           (not safe)
3.           (inlining-limit 0))
4.
5.  (define (foreach f lst)
6.    (let loop ((lst lst))
7.      (if (pair? lst)
8.          (begin
9.            (f (car lst))
10.           (loop (cdr lst)))
11.         #f)))
```

Figure 1. Source code of the `foreach` function

The GVM code generated for this example is given in Figure 2 (the code's syntax has been altered in minor ways from the normal compiler output to make it easier to follow). In the GVM code small integers prefixed with a "#" are basic block labels. The front-end has translated the call to `pair?` into an *ifjump* instruction of the primitive `##pair?`. It has also translated the calls to `car` and `cdr` into *apply* instructions of the primitives `##car` and `##cdr` respectively, which do not check the type of their argument.

Upon entry to the `foreach` function, at basic block #1, the parameters `f` and `lst` are contained in `r1` and `r2` respectively, and `r0` contains the return address. When the list `lst` is non-empty, all three registers are saved to the stack (at lines 14-16) to create a continuation frame for the non-tail call to `f`. `r1` is set to the first element of the list, `r0` is set to the return point, a reference to basic block #2, and `f` is jumped to (at line 22) with an argument count of 1 and a frame size of 3 to account for the allocation of the continuation frame and an empty activation frame. At the return point, basic block #2, the continuation frame is read (at lines 5-7) to prepare the tail call to `loop` (at line 8). The tail call is to a known

```
1.  #1 fs=0 entry-point nargs=2
2.    jump fs=0 #3
3.
4.  #2 fs=3 return-point
5.    r2 = (##cdr frame[3])
6.    r1 = frame[2]
7.    r0 = frame[1]
8.    jump/poll fs=0 #3
9.
10. #3 fs=0
11.   if (##pair? r2) jump fs=0 #4 else #6
12.
13. #4 fs=0
14.   frame[1] = r0
15.   frame[2] = r1
16.   frame[3] = r2
17.   r1 = (##car r2)
18.   r0 = #2
19.   jump/poll fs=3 #5
20.
21. #5 fs=3
22.   jump fs=3 frame[2] nargs=1
23.
24. #6 fs=0
25.   r1 = '#f
26.   jump fs=0 r0
```

Figure 2. GVM code generated for the `foreach` function

function so it is simply a jump to basic block #3 with a frame size of 0 to account for the deallocation of the continuation frame.

Finally, note the placement of two interrupt checks at lines 8 and 19 which guarantee a bounded number of GVM instructions executed between interrupt checks.

3. Translation to JavaScript

We will explain the translation process by referring to the final JavaScript code produced when compiling the `foreach` function. Figure 3 gives the relevant parts of the code.

To avoid name clashes with other code, all JavaScript global variables and function names are prefixed by "`Gambit_`" in the code actually generated by the compiler. For presentation purposes, we have stripped this prefix and made some minor syntactic changes (such as removing redundant braces). Some optimizations which are discussed in Section 3.4 have also been disabled to improve readability.

3.1 GVM State

Efficient access to the GVM state is critical to achieve good execution speed. For this reason the GVM state is stored in JavaScript global variables (lines 1-5 in Figure 3). The stack and global variables are implemented with JavaScript arrays. Note that JavaScript arrays grow automatically when storing beyond the last element, which is convenient for implementing a stack. The registers, stack pointer and argument count are also JavaScript global variables.

3.2 Data Representation

When possible, Scheme types are mapped to similar JavaScript types. For example Booleans to JavaScript Booleans, vectors to JavaScript arrays, and the empty list to JavaScript's `null`.

Some types, such as pairs, strings and characters are JavaScript objects with their own constructors (for example the constructor for pairs is at lines 7-10). Strings can't be mapped to JavaScript strings which are immutable. However, symbols and keywords are mapped to JavaScript strings.

```
1.  var reg0, reg1, reg2, reg3, reg4; // registers
2.  var stack = [false]; // runtime stack
3.  var glo = {};        // Scheme global variables
4.  var sp = 0;          // stack pointer
5.  var nargs;           // argument count
6.
7.  function Pair(car, cdr) {
8.    this.car = car;
9.    this.cdr = cdr;
10. }
11.
12. function run(pc) {
13.   while (pc !== false)
14.     pc = pc();
15. }
16.
17. function bb1_foreach() { // entry-point
18.   if (nargs !== 2)
19.     return wrong_nargs(bb1_foreach);
20.   return bb3_foreach;
21. }
22. bb1_foreach.id = "bb1_foreach"; // meta info
23.
24. function bb3_foreach() {
25.   if (reg2 instanceof Pair) {
26.     stack[sp+1] = reg0;
27.     stack[sp+2] = reg1;
28.     stack[sp+3] = reg2;
29.     reg1 = reg2.car;
30.     reg0 = bb2_foreach;
31.     sp += 3;
32.     return poll(bb5_foreach);
33.   } else {
34.     reg1 = false;
35.     return reg0;
36.   }
37. }
38.
39. function bb2_foreach() { // return-point
40.   reg2 = stack[sp].cdr;
41.   reg1 = stack[sp-1];
42.   reg0 = stack[sp-2];
43.   sp += -3;
44.   return poll(bb3_foreach);
45. }
46. bb2_foreach.id = "bb2_foreach"; // meta info
47. bb2_foreach.fs = 3;
48. bb2_foreach.link = 1;
49.
50. function bb5_foreach() {
51.   nargs = 1;
52.   return stack[sp-1];
53. }
```

Figure 3. JavaScript code generated for the `foreach` function

In order to implement the full numeric tower, different concrete types are used to implement numbers. Fixnums are mapped to JavaScript numbers, and bignums, flonums, etc are JavaScript objects with specific constructors.

Functions, whether they are closures or not, are mapped to JavaScript functions. However, because the function call protocol uses the GVM registers and stack to pass arguments, the JavaScript functions are parameterless. For example, the Scheme `foreach` function is implemented by the JavaScript `bb1_foreach` function at line 17.

3.3 Basic CFG Translation

If we discount the branch destination inlining optimization which is explained in the next section, the back-end translates each basic block to a parameterless JavaScript function. Most GVM instructions are translated straightforwardly to JavaScript code. The branch instruction at the end of the basic block is translated to a `return` statement which returns the destination operand, that is a reference to the JavaScript function containing the code of the destination basic block, or a JavaScript closure (see Section 3.6).

For example, the GVM branch instruction at the end of basic block #1 is translated at line 20 to a return of a reference to function `bb3_foreach` which corresponds to basic block #3.

A trampoline, implemented by the function `run` at line 12, is used to sequence the flow of control from the source to destination basic blocks. The program is started by calling `run` with a reference to the basic block of the program's entry point.

The `poll` function called at lines 32 and 44 is responsible for interrupt handling. After checking for interrupts and invoking the appropriate handler, the `poll` function returns its argument.

3.4 Optimizations

With the basic translation each GVM branch incurs the run time cost of one function return and call. The cost of the trampoline and interrupt checks is reduced using the following optimizations:

Branch destination inlining. Basic blocks which are only referenced in a single branch instruction or are very short (only contain a branch instruction) are inlined at the location of the branch. This happens frequently in *ifjump* instructions, effectively recovering in the target language some of the structure of the source `if`. For example, the destination basic blocks #4 and #6 have been inlined in the `if` at line 25.

Branch destination call. Instead of returning the destination operand to the trampoline, it is possible to return the result of calling the destination operand. For example, the branch to basic block #3 at line 20 is really implemented with `return bb3_foreach();`. This makes it possible for the JavaScript VM to optimize the control flow and perhaps inline the body of the destination function. There will be an accumulation of stack frames on the JavaScript VM if it doesn't do tail call optimization. However, the depth of the stack is bounded because of the presence of the calls to `poll`, which cause an unwind of the VM's stack all the way back to the trampoline.

Intermittent polling. The frequency of calls to the `poll` function is reduced by using a counter. Each branch instruction with a poll flag decrements the counter. When it reaches 0, the `poll` function is called, and the counter is reset (to 100). For example, line 32, which is a polling branch to basic block #5, is really implemented with the code:

```
if (--poll_count === 0)
  return poll(bb5_foreach);
else
  return bb5_foreach();
```

3.5 Meta Information

The code generated also stores some meta information on the first-class basic blocks (functions `bb1_foreach` and `bb2_foreach`). The property `id` set at lines 22 and 46 is required for serialization of Scheme functions and continuations. For the return point basic block #2 the properties `fs` and `link` are set at lines 47 and 48. This is required for the implementation of continuations and is further discussed in Section 4.

3.6 Closures

The mapping from Scheme closures to JavaScript closures is designed to support closure serialization. The GVM's flat closures

are composed of a number of slots, including a slot referring to the closure entry point. The slots of a Scheme closure are implemented as properties of the JavaScript closure. The only closed variable of the JavaScript closure is a reference to itself.

Consider the `ccons` function (curried `cons`) whose definition is given in Figure 4 and whose generated JavaScript code is in Figure 5.

```
1.  (define (ccons x)
2.    (lambda (y) (cons x y)))
```

Figure 4. Source code of the `ccons` function

```
1.  function closure_alloc(entry_bb) {
2.
3.    function self() {
4.      reg4 = self;
5.      return self.v0;
6.    }
7.
8.    self.v0 = entry_bb;
9.
10.   return self;
11. }
12.
13. function bb1_ccons() { // entry-point
14.   if (nargs !== 1)
15.     return wrong_nargs(bb1_ccons);
16.   var closure1 = closure_alloc(bb2_ccons);
17.   stack[sp+1] = closure1;
18.   closure1.v1 = reg1;
19.   reg1 = stack[sp+1];
20.   return reg0;
21. }
22. bb1_ccons.id = "bb1_ccons"; // meta info
23.
24. function bb2_ccons() { // closure-entry-point
25.   if (nargs !== 1)
26.     return wrong_nargs(bb2_ccons);
27.   reg4 = reg4.v1;
28.   reg1 = new Pair(reg4, reg1);
29.   return reg0;
30. }
31. bb2_ccons.id = "bb2_ccons"; // meta info
```

Figure 5. JavaScript code generated for the `ccons` function

The construction of a Scheme closure is a two step process. First, it is allocated using the `closure_alloc` function (line 1). The actual JavaScript closure is the `self` function defined at line 3. Then its slots are initialized using property assignments (line 18).

The property `v0` of the JavaScript closure is set to the closure's entry point (line 8). When the closure is called, the property `v0` of the closure is accessed (line 5) to branch to the correct closure entry point. `r4` will have been set to a reference to the closure itself (line 4), so that access to closed variables is possible. For example the access to `x` is translated to reading property `v1` of the closure (line 27).

4. Implementing Continuations

4.1 Continuation Management

We use the *incremental stack/heap approach* for managing continuations [7]. This approach allows the GVM code to use a standard function call protocol.

In the incremental stack/heap approach, the continuation, which is conceptually a list of continuation frames, is stored in the stack and in the heap. The more recent continuation frames are stored in the stack, and older continuation frames form a linked chain of objects (as in the "before" part of Figure 7 which has 3 frames in the stack, and one in the heap). The continuation frames in the stack are not explicitly linked, but those in the heap are.

Continuation frames are initially allocated in the stack, and in some cases, such as when the current continuation is reified by `call/cc`, they are later copied to the heap. The process of copying the stack frames to the heap is called *continuation heapification*. For this it is necessary to find where each stack frame starts and ends by parsing all the stack. This is achieved by attaching meta information to each return point: the continuation frame size (`fs`), and the index of the slot in that frame where the return address is stored (`link`). For example, the continuation frame created for the non-tail call to `f` in the `foreach` has `fs`=3 and `link`=1 (this meta information is set at lines 47-48 in Figure 2).

Given a stack of continuation frames, and the current return address (`ra`), it is a simple matter to iterate over the frames from newest to oldest. The topmost frame has a size of `ra.fs`, and `stack[sp - ra.fs + ra.link]` is the return address in that frame, which can be used to parse the next stack frame. This process is repeated until the base of the stack is reached.

Each continuation frame in the heap is represented as a JavaScript array with one more element than the frame size. If we call `ra` the return address attached to the frame `frm`, then `frm[0]` contains `ra` and `frm[ra.link]` contains the next frame in the chain (the value `false` indicates the end of the chain). In other words, the heap frames are chained using the slot of the frame which normally contains the return address. All other slots of the continuation frame are stored in the corresponding index in the array.

In our implementation, we store in `stack[0]` the reference to the most recent continuation frame in the heap (the first in the chain). The oldest continuation frame in the stack, which starts at `stack[1]`, is a special frame because the return address it contains is always the function `underflow`. When the function which created that frame returns, the frame will be deallocated, making the stack empty, and control will be transferred to the `underflow` function. This function causes the heap frame in `stack[0]` to be copied to the stack and control is transferred to that frame's return address. In order to prepare for the next time the stack is emptied, a reference to the next heap frame is copied to `stack[0]`, and the slot of the stack frame which contains the return address is set to the function `underflow`. The definition of the `underflow` function is given in Figure 6.

```
1.  function underflow() {
2.
3.    var frm = stack[0];
4.
5.    if (frm === false) // end of continuation?
6.      return false;    // terminate trampoline
7.
8.    var ra = frm[0];
9.    var fs = ra.fs;
10.   var link = ra.link;
11.   stack = frm.slice(0, fs + 1);
12.   sp = fs;
13.   stack[0] = frm[link];
14.   stack[link] = underflow;
15.
16.   return ra;
17. }
```

Figure 6. Definition of the `underflow` function

```
1.  function heapify(ra) {
2.
3.    var chain = false;
4.    var prev_frm = false;
5.    var prev_link;
6.
7.    while (sp !== 0) { // stack not empty
8.      var fs = ra.fs;
9.      var link = ra.link;
10.     var frm = stack.slice(sp - fs, sp + 1);
11.     if (prev_frm === false)
12.       chain = frm;
13.     else
14.       prev_frm[prev_link] = frm;
15.     prev_frm = frm;
16.     frm[0] = ra;
17.     sp = sp - fs;
18.     ra = stack[sp + link];
19.     prev_link = link;
20.   }
21.
22.   if (prev_frm === false)
23.     chain = stack[0];
24.   else
25.     prev_frm[prev_link] = stack[0];
26.
27.   stack = [chain];
28.   sp = 0;
29.
30.   return underflow;
31. }
```

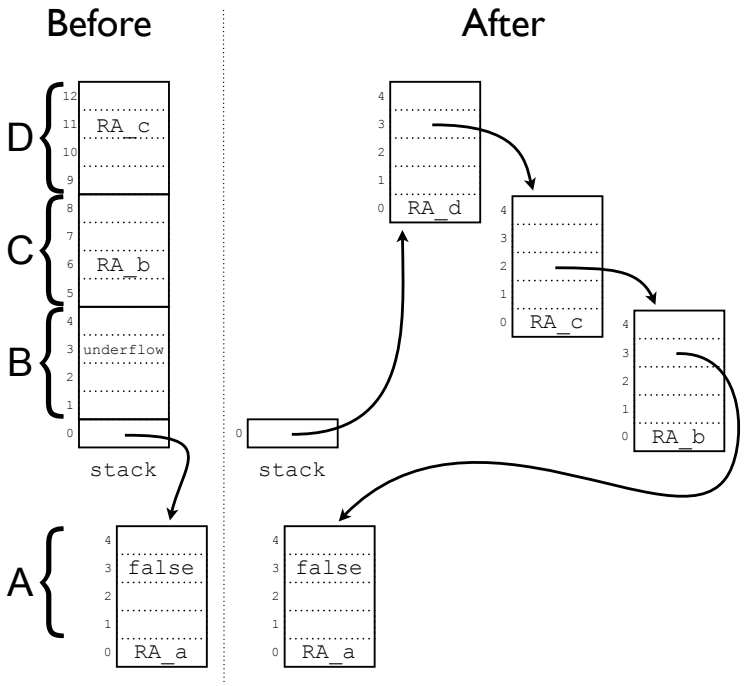

Figure 7. Continuation heapification algorithm and example. Before heapification, continuation frames B, C and D are on the stack. After heapification with the call `heapify(RA_d)`, where `RA_d` is the return address back to the function which created frame D, all frames are in the heap and explicitly linked using the frame slot normally containing the return address.

```
1.  function bb1_continuation_capture() {
2.    if (nargs !== 1)
3.      return wrong_nargs(bb1_continuation_capture);
4.    var receiver = reg1;
5.    reg0 = heapify(reg0);
6.    reg1 = stack[0];
7.    nargs = 1;
8.    return receiver;
9.  }
10.
11. function bb1_continuation_return() {
12.   if (nargs !== 2)
13.     return wrong_nargs(bb1_continuation_return);
14.   sp = 0;
15.   stack[0] = reg1;
16.   reg0 = underflow;
17.   reg1 = reg2;
18.   return reg0;
19. }
```

Figure 8. Implementation of `continuation-capture` and `continuation-return`.

4.2 Continuation Heapification and `call/cc`

Continuation heapification is implemented with the function `heapify` given in Figure 7. The parameter `ra` is the return address back to the function which created the topmost continuation frame.

With the `heapify` function, it is easy to implement the continuation API of [7]. The functions `continuation-capture` and `continuation-return` are implemented by the JavaScript functions given in Figure 8. The `call/cc` function is then implemented with these functions using:

```
(define (call/cc receiver)
  (continuation-capture
   (lambda (k)
     (receiver (lambda (r)
                 (continuation-return k r))))))
```

5. Evaluation

5.1 Goal and Methodology

In this section we aim to evaluate the performance of the three approaches discussed in this paper and which are implemented in the Gambit-JS, Scheme2JS and Spock compilers. Our methodology consists in executing with each system specially selected benchmark programs which represent use-cases of tail calls and continuations.

Although it has the virtue of being empirical, the methodology has pitfalls for the comparison of the approaches because the compilers may adopt different implementation strategies for features unrelated to tail calls and first-class continuations. Some optimization may be implemented in one compiler and not the other, even though it could have been, which gives one compiler an advantage that is not related to the continuation implementation approach. We are interested here in comparing the approaches, not the compilers. For this reason we have carefully chosen the source programs, programming style, declarations, and command-line options, to avoid unrelated differences. The target JavaScript code generated was ex-

amined manually to ensure performance differences were mainly due to the continuation implementation approach. Specifically, we have avoided:

Local definitions. The Scheme2JS compiler is able to translate the parts of the source program almost directly into the isomorphic JavaScript code when it can determine that first-class continuations need not be supported for those parts. This is frequently the case when the entire benchmark program is a set of definitions within an enclosing function (because the program analysis is simpler). The other compilers do not have this optimization.

Non-primitive library functions. Primitive library functions like `cons` and `car` are implemented similarly by the different compilers and are inlined. More complex library functions, such as `append`, `map` and `equal?`, have a wider range of possible implementations (level of type checking, precision of error messages, variation in object representation, etc). For this reason, benchmarks which use non-primitive library functions contain a generic Scheme definition of the function which uses primitive library functions.

Type checking. Scheme2JS and Spock primitive functions do not type check their arguments. Gambit-JS's type checking was disabled with the declaration `(declare (not safe))`.

Non-integer numbers. Scheme2JS and Spock use JavaScript numbers to represent Scheme numbers, which means that they have a partial implementation of the numeric tower. The declaration `(declare (fixnum))` was used for Gambit-JS so that all arithmetic operations would be performed on JavaScript numbers, like the other systems.

Function inlining. The compilers do user function inlining differently and under different conditions. Because function inlining has a big impact on performance, it has been disabled with Gambit's `(declare (inlining-limit 0))` declaration and Scheme2JS's command line option `--max-inline-size 0`. Spock does not inline functions.

Scheme2JS and Spock do not perform argument count checking because they use the JavaScript semantics for argument passing where it is allowed to pass fewer or more arguments than there are formal parameters. Gambit-JS does perform argument count checking as it is necessary for rest parameter handling, and it provides additional safety and precise error messages. It is not easy to remove the argument count checking in general, and it can be argued that it is consistent with the virtual machine approach, so it was not disabled in the experiments. The overhead of argument count checking is fairly low (we have measured experimentally using `fib` that the overhead is less than 5%).

5.2 Benchmark Programs

There are two groups of benchmark programs. The first group, containing the programs `fib35`, `nqueens12`, and `oddeven`, do not manipulate first-class continuations. The purpose of these programs is to evaluate the impact on function calls of supporting first-class continuations. The program `oddeven` performs only tail calls.

The second group use `call/cc` in various ways. The programs `ctak` and `contfib30` have non-tail recursive functions: `ctak` reifies each continuation of its recursion, and `contfib30` reifies only the continuations at the leaves of the recursion. The program `btsearch2000` performs a backtracking search, and `threads10` is a thread scheduler which interleaves the execution of 10 threads.

The source code of the benchmark programs is given in Appendix A.

5.3 Setting

The V8 JavaScript VM version 3.4.3, in its command-line variant `d8`, running on a OS X 10.8 computer with a 2.2 GHz Intel Core i7 processor and 16 GB RAM is used in all the experiments. The Scheme systems used are:

- Gambit-JS version v4.6.6 20120811162045 with the declarations `(declare (standard-bindings) (fixnum) (not safe) (inlining-limit 0))`,
- Scheme2JS version 20110717 with command-line options: `--max-inline-size 0 --call/cc --trampoline`,
- Spock version 4.7.0 with no special command-line options.

5.4 Results

The execution times of the benchmark programs are given in Figure 9.

Program	Gambit-JS	Scheme2JS		Spock	
`fib35`	1.038	3.273	(3.2)	3.226	(3.1)
`nqueens12`	.907	1.450	(1.6)	3.109	(3.4)
`oddeven`	.851	2.419	(2.8)	7.132	(8.4)
`ctak`	.251	35.048	(139.6)	.400	(1.6)
`contfib30`	1.503	198.718	(132.2)	2.088	(1.4)
`btsearch2000`	1.449	39.723	(27.4)	4.267	(2.9)
`threads10`	1.606	39.914	(24.8)	3.459	(2.2)

Figure 9. Execution times of the benchmark programs. The time is given in seconds. The number in parentheses are the times relative to Gambit-JS.

On the benchmarks, the VM approach (Gambit-JS) is consistently faster than the Replay-C algorithm (Scheme2JS) and CPS conversion (Spock). It is 1.4 to 8.4 times faster than CPS conversion, and 1.6 to 139.6 times faster than the Replay-C algorithm.

The Replay-C algorithm has its best relative times when `call/cc` is not used (1.6 to 3.2 times slower than the VM approach). When `call/cc` is used, the performance depends greatly on the depth of recursion (24.8 to 139.6 times slower than the VM approach). This is because the Replay-C algorithm restores the complete continuation on the JavaScript VM stack every time a continuation is invoked. Our approach restores continuations incrementally, one frame at a time, so the cost does not depend on the depth of the continuation.

CPS conversion makes it trivial to reify continuations because all functions are passed an explicit continuation parameter. Unsurprisingly, the CPS conversion approach has its best relative times when `call/cc` is used (1.4 to 2.9 times slower than the VM approach). When `call/cc` is not used the relative times range from 3.1 to 3.4 when non-tail calls are performed. This is an indication that the creation of closures for the continuation frames of non-tail calls is more expensive than using an explicit representation on a stack. It is surprising that for `oddeven`, which only performs tail calls (i.e. no continuation frames are created), the relative time goes up to 8.4. This is probably due to the cost of unwinding the JavaScript VM's stack at regular intervals to avoid overflowing it. Spock does this through a check at every function entry, similar to Gambit-JS's interrupt checks, but not intermittently. When a counter is added to check intermittently, the time drops to 3.767 seconds, which is still 4.4 times slower than Gambit-JS. It is likely that this high cost is accounted for by a bad interaction between the structure of the generated code and the V8 optimizer (in particular the Spock stack checks use the JavaScript `arguments` form, which can disable some optimizations).

6. Other Target Languages

In order to implement the python and ruby backend, we use native types where applicable and define specific objects in order to preserve scheme's semantic properties in the targeted language.

Scheme / GVM	Python	Comment
fixnum	native	
float	native	
complex	native	
bignum	native	
char	custom type	
string	custom type	mutable strings
symbol	custom type	using native string
keyword	custom type	using native string
vector	native	
pair	custom type	
'()	None	
#t	True	
#f	False	
eq?	is	
equal?	==	
closure and slots	native function and attributes	See Figure 10
continuation	custom type	
stack	dict	
registers	dict	or global variable
global var	dict	

Table 1. Implementations details

```
1.  def closure_alloc(entry_bb):
2.
3.    def self():
4.      global reg4
5.      reg4 = self
6.      return self.v0
7.
8.    self.v0 = bb
9.    return self
```

Figure 10. Python `closure_alloc` function

7. Conclusion

We have proposed a VM-based approach for implementing tail calls and first-class continuations in dynamic languages which do not have those features. Our approach compiles Scheme source programs into an intermediate language, the Gambit Virtual Machine (GVM), which is then translated to the target language using a trampoline and an explicit representation of the GVM runtime stack. This allows continuations to be implemented with most of the algorithms used by native code compilers [5]. We use the incremental stack/heap approach [5] which allows the GVM code to use a standard function call protocol, with a zero overhead for code which doesn't manipulate first-class continuations, and which has a cost for invoking a continuation which is proportional to the size of the topmost continuation frame.

Our experiments on specially selected benchmark programs in the context of JavaScript show that the approach compares favorably to the Replay-C algorithm used in the Scheme2JS compiler and to the CPS conversion used in the Spock compiler. The execution time is consistently faster for the VM-based approach, up to two orders of magnitude for programs using `call/cc` intensely in non-tail recursive functions.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada and Mozilla Corporation.

References

[1] URL `http://www.call-cc.org/`.

[2] URL `https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS/`.

[3] URL `http://wiki.call-cc.org/eggref/4/spock`.

[4] H. G. Baker. Cons should not cons its arguments, part ii: Cheney on the m.t.a. *SIGPLAN Not.*, 30(9):17–20, Sept. 1995. ISSN 0362-1340. doi: 10.1145/214448.214454. URL `http://doi.acm.org/10.1145/214448.214454`.

[5] W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher Order Symbol. Comput.*, 12(1):7–45, Apr. 1999. ISSN 1388-3690. doi: 10.1023/A:1010016816429. URL `http://dx.doi.org/10.1023/A:1010016816429`.

[6] M. Feeley. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 179–187, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: 10.1145/165180.165205. URL `http://doi.acm.org/10.1145/165180.165205`.

[7] M. Feeley. A better api for first-class continuations, 2001.

[8] M. Feeley and J. S. Miller. A parallel virtual machine for efficient scheme compilation. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 119–130, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: 10.1145/91556.91606. URL `http://doi.acm.org/10.1145/91556.91606`.

[9] G. Germain, M. Feeley, and S. Monnier. Concurrency oriented programming in termite scheme. 2005.

[10] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. 25(6):66–77, June 1990. ISSN 0362-1340. doi: 10.1145/93548.93554. URL `http://doi.acm.org/10.1145/93548.93554`.

[11] F. Loitsch. Javascript to Scheme compilation. In *Proceedings of the Sixth Workshop on Scheme and Functional Programming*, pages 101–116, Sept. 24, 2005.

[12] F. Loitsch. Exceptional continuations in JavaScript. In *2007 Workshop on Scheme and Functional Programming*, Sept. 2007.

[13] F. Loitsch. *Scheme to JavaScript Compilation*. PhD thesis, Université de Nice - Sophia Antipolis, Mar. 2009.

[14] M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Proceedings of the Second International Symposium on Static Analysis*, SAS '95, pages 366–381, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60360-3. URL `http://dl.acm.org/citation.cfm?id=647163.717691`.

[15] D. Yoo. *Building Web Based Programming Environments for Functional Programming*. PhD thesis, Worcester Polytechnic Institute, feb 2012. URL `http://www.wpi.edu/Pubs/ETD/Available/etd-042612-104736/`.

A. Source Code of Benchmark Programs

```
1.  (define (fib n)
2.    (if (< n 2)
3.        1
4.        (+ (fib (- n 1))
5.           (fib (- n 2)))))
6.
7.  (run-benchmark
8.   "fib35"
9.   (lambda () (fib 35)))
```

Figure 11. Source code of `fib35`.

```
(define (app lst1 lst2)
  (if (pair? lst1)
      (cons (car lst1) (app (cdr lst1) lst2))
      lst2))

(define (one-up-to n)
  (let loop ((i n) (lst '()))
    (if (= i 0)
        lst
        (loop (- i 1) (cons i lst)))))

(define (explore x y placed)
  (if (pair? x)
      (+ (if (ok? (car x) 1 placed)
             (explore (app (cdr x) y)
                      '()
                      (cons (car x) placed))
             0)
         (explore (cdr x)
                  (cons (car x) y)
                  placed))
      (if (pair? y) 0 1)))

(define (ok? row dist placed)
  (if (pair? placed)
      (and (not (= (car placed) (+ row dist)))
           (not (= (car placed) (- row dist)))
           (ok? row (+ dist 1) (cdr placed)))
      #t))

(define (nqueens n)
  (explore (one-up-to n)
           '()
           '()))

(run-benchmark
 "nqueens12"
 (lambda () (nqueens 12)))
```

Figure 12. Source code of `nqueens12`.

```
(define (odd n)
  (if (= n 0) #f (even (- n 1))))

(define (even n)
  (if (= n 0) #t (odd (- n 1))))

(run-benchmark
 "oddeven"
 (lambda () (odd 100000000)))
```

Figure 13. Source code of `oddeven`.

```
(define (ctak x y z)
  (call-with-current-continuation
   (lambda (k) (ctak-aux k x y z))))

(define (ctak-aux k x y z)
  (if (not (< y x))
      (k z)
      (ctak-aux
       k
       (call-with-current-continuation
        (lambda (k) (ctak-aux k (- x 1) y z)))
       (call-with-current-continuation
        (lambda (k) (ctak-aux k (- y 1) z x)))
       (call-with-current-continuation
        (lambda (k) (ctak-aux k (- z 1) x y))))))

(run-benchmark
 "ctak"
 (lambda () (ctak 22 12 6)))
```

Figure 14. Source code of `ctak`.

```
(define (contfib n)
  (if (< n 2)

      (call-with-current-continuation
       (lambda (k)
         (k 1)))

      (+ (contfib (- n 1))
         (contfib (- n 2)))))

(run-benchmark
 "contfib30"
 (lambda () (contfib 30)))
```

Figure 15. Source code of `contfib30`.

```
(define fail (lambda () #f))

(define (in-range a b)
  (call-with-current-continuation
   (lambda (cont)
     (enumerate a b cont))))

(define (enumerate a b cont)
  (if (> a b)
      (fail)
      (let ((save fail))
        (set! fail
          (lambda ()
            (set! fail save)
            (enumerate (+ a 1) b cont)))
        (cont a))))

(define (btsearch n)
  (let* ((n*2 (* n 2))
         (x (in-range 0 n))
         (y (in-range 0 n)))
    (if (< (+ x y) n*2)
        (fail) ;; backtrack
        (cons x y))))

(run-benchmark
 "btsearch2000"
 (lambda () (btsearch 2000)))
```

Figure 16. Source code of `btsearch2000`.

```
;; Queues.

(define (next q) (vector-ref q 0))
(define (prev q) (vector-ref q 1))
(define (next-set! q x) (vector-set! q 0 x))
(define (prev-set! q x) (vector-set! q 1 x))

(define (empty? q) (eq? q (next q)))

(define (queue) (init (vector #f #f)))

(define (init q)
  (next-set! q q)
  (prev-set! q q)
  q)

(define (deq x)
  (let ((n (next x)) (p (prev x)))
    (next-set! p n)
    (prev-set! n p)
    (init x)))

(define (enq q x)
  (let ((p (prev q)))
    (next-set! p x)
    (next-set! x q)
    (prev-set! q x)
    (prev-set! x p)
    x))

;; Process scheduler.

(define (boot)
  ((call-with-current-continuation
    (lambda (k)
      (set! graft k)
      (schedule)))))

(define graft #f)
(define current #f)
(define readyq (queue))

(define (process cont)
  (init (vector #f #f cont)))

(define (cont p) (vector-ref p 2))
(define (cont-set! p x) (vector-set! p 2 x))

(define (spawn thunk)
  (enq readyq
       (process (lambda (r)
                  (graft (lambda ()
                           (end (thunk))))))))

(define (schedule)
  (if (empty? readyq)
      (graft (lambda () #f))
      (let ((p (deq (next readyq))))
        (set! current p)
        ((cont p) #f))))

(define (end result) (schedule))

(define (yield)
  (call-with-current-continuation
   (lambda (k)
     (cont-set! current k)
     (enq readyq current)
     (schedule))))

(define (wait x)
  (if (> x 0)
      (begin
        (yield)
        (wait (- x 1)))))

(define (threads n)

  (let loop ((n n))
    (if (> n 0)
        (begin
          (spawn (lambda () (wait 100000)))
          (loop (- n 1)))))

  (boot))

(run-benchmark
 "threads10"
 (lambda () (threads 10)))
```

Figure 17. Source code of `threads10`.