Compiling for Multi-language Task Migration

Marc Feeley
Université de Montréal, Canada
feeley@iro.umontreal.ca

Abstract

Task migration allows a running program to continue its execution in a different destination environment. Increasingly, execution environments are defined by combinations of cultural and technological constraints, affecting the choice of host language, libraries and tools. A compiler supporting multiple target environments and task migration must be able to marshal continuations and then unmarshal and continue their execution, ideally, even if the language of the destination environment is different. In this paper, we propose a compilation approach based on a virtual machine that strikes a balance between implementation portability and efficiency. We explain its implementation within a Scheme compiler targeting JavaScript, PHP, Python, Ruby and Java – some of the most popular host languages for web applications. As our experiments show, this approach compares well with other Scheme compilers targeting high-level languages in terms of execution speed, being sometimes up to 3 orders of magnitude faster.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Compilers

Keywords Task migration, continuation, tail call, marshalling, Scheme, JavaScript, trampoline, virtual machine

1. Introduction

High-level programming languages are increasingly being used as targets for compiling other source languages [1, 3, 11]. Some of the more popular target languages are C, Java and JavaScript. Such an approach gives increased portability, allowing the source language to execute wherever the target language can be executed, it gives access to the features available in the target language (libraries, tools, etc), and it simplifies integrating program parts written in the source language with an existing code base in the target language.

Compilers supporting multiple target languages are attractive when the same software must execute in multiple environments where specific host programming languages are a requirement for technological or cultural reasons (for example JavaScript in a web browser and PHP in a web server). Implementing a multi-target compiler for a distributed programming language supporting task migration, such as CmPS [13], Kali [4] and Termite Scheme [10], places extra constraints on the compilation approach.

As an example using Termite Scheme, consider a task running on a web browser that wants to migrate to a designated *destination* web browser on a different computer. This could be achieved by migrating the task to the web server and then migrating the task again to the destination web browser connected to that web server. In Termite Scheme these steps can be expressed concisely with the following function definition:

```
(define (migrate-to-other-browser destination)
  (migrate-task web-server)
  (migrate-task destination))
```

The `migrate-task` function is defined by the Termite Scheme runtime system as follows:

```
(define (migrate-task node)
  (call/cc (lambda (k)
             (remote-spawn node (lambda () (k #t)))
             (halt!))))
```

The call to `remote-spawn` sends its second argument, a closure containing `k`, the continuation of the call to `migrate-task`, to the destination computational node (the web server or destination web browser) where a thread is started which calls the closure, thus resuming at the destination the computation following the call to `migrate-task` (the continuation `k`). The thread at the source node is terminated by the call to `halt!`.

Consequently, for implementing this task migration approach, it must be possible to capture a running computational task's continuation, marshal it and then unmarshal and continue its execution, ideally, regardless of the destination environment's host language. This would allow, for example, the migration of a running task between a web browser (hosted on JavaScript) and a web server (say hosted

DLS'15, October 27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3690-1/15/10...$15.00
http://dx.doi.org/10.1145/2816707.2816713

on PHP). This paper proposes a compilation approach that supports task migration between the target languages of a Scheme [14, 19] compiler supporting Termite Scheme [10].

Using dynamic languages such as JavaScript and PHP as target languages for a Scheme compiler is alluring because they offer dynamic typing, introspective features, closures and garbage collection that simplify the translation of Scheme. The biggest challenge remaining is the implementation of tail calls and first-class continuations, which have no direct equivalent in general in these target languages.

Scheme systems conforming to the standard must implement tail calls without stack growth. Contrary to most other languages, in Scheme this behavior is a requirement, not an optional optimization. A Scheme system must also implement the `call/cc` primitive 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 by using loops, and one-shot escape continuations by 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 targeting high-level languages such as the Bigloo [18], Chicken [21] and Gambit-C [8] Scheme to C compilers, and the Scheme2JS [15–17], Spock [20], and Whalesong [22] 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 [2], 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. By using a CPS conversion of the code, the Cheney on the MTA approach makes it possible to reclaim all stack frames because all calls are tail calls in the CPS'ed code. The implementation of first-class continuations is greatly simplified since all functions receive an explicit continuation function.

In Scheme2JS, first-class continuations are implemented by copying the stack frames to the heap. In 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 [16].

Gambit-C uses a virtual machine based representation of the program. The instructions of this virtual machine are translated to C in a fairly direct way. The virtual machine models the stack explicitly as a C array, and consequently could implement first-class continuations with most of the algorithms used by native code compilers [5]. It uses a fine grained variant of the Hieb-Dybvig-Bruggeman strategy [12]. In this paper we use a similar virtual machine based compilation approach but targeting multiple languages, specifically Java, JavaScript, PHP, Python and Ruby, which are popular languages for web development.

These approaches to implementing tail calls and first-class continuations offer different trade-offs. It is convenient to consider as a baseline a direct translation of Scheme to the target language that ignores tail calls and first-class continuations, and to look at the overhead over the baseline for a particular approach.

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 it uses has more work to do when first-class continuations are captured and invoked, for copying between the stack and heap. Programs that seldom capture and invoke continuations still have an overhead for the `try/catch` forms that wrap all non-tail calls, but it is possibly less work than creating and reclaiming continuation closures, 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 local variables are converted, in the general case, to target language array indexing operations of the stack, preventing an efficient assignment of the variables to registers. All the approaches generate code with a more complex structure than the baseline. This also causes overhead because optimization by the target VM is hindered.

In this paper we explore an approach for implementing tail calls and first-class continuations that is based on a virtual machine and that supports marshalling and unmarshalling function closures and continuations. The approach strikes a good balance between simplicity of implementation and performance. A performance analysis of the JavaScript back-end reveals that Gambit generates code that is consistently faster than both Scheme2JS and Spock on the four JavaScript VMs we have tested, and on some JavaScript VMs the performance gap is substantial.

2. Gambit Virtual Machine

The Gambit compiler front-end follows a fairly standard organization as a pipeline of stages that 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 [9], called the Gambit Virtual Machine (GVM). The GVM is a simple machine with a set of locations into 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 that 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`). The stack is only used when there are more than 3 arguments. Register `r4` is used to implement closures, as explained below.

There are seven GVM instructions: *label*, *jump*, *ifjump*, *switch*, *copy*, *apply*, and *close*. Each basic block begins with a *label* instruction that identifies the basic block, gives its kind, and the frame size of the topmost stack frame. There are local blocks, used for control flow between blocks of a function, and first-class blocks, used for control flow between functions (function entry-point and 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 that transfers control to another basic block 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 specifying the argument count. The *label* instruction for the entry-point specifies 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 that 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 specify the destination GVM location, and the source operands, which can be any GVM location, immediate Scheme object or reference to a first-class block. Conceptually, the *copy* instruction is equivalent to an *apply* instruction of the identity function, but they are kept separate for historical reasons.

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. In the function's body, 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 of stack frames is specified implicitly in the *label* and branch instructions. The *label* instruction indicates the topmost frame's size immediately after its execution (the `fs=`n annotation). 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 is 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 branch instruction. Moreover, the back-end can use the entering 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 that 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

```
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

stack area overflow, heap overflow, and preemptive multi-threading timer interrupt, that disrupts the normal sequence of execution. The GVM polls for interrupts using *interrupt checks* that 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_{max}, the maximum poll latency (see [6] for details). Consequently, the bounds of the stack area will never be exceeded if an extra L_{max} slots are reserved at the end of the stack 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.

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.

Basic blocks #1 and #2 are first-class blocks (a function entry-point and return-point respectively) and the others are local blocks. Upon entry to the `foreach` function, at basic

```
1.  #1 fs=0 entry-point nparams=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

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 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 that guarantee a bounded number of GVM instructions executed between interrupt checks and consequently stack-overflow checks.

3. Data Representation

The main difficulty in translating GVM code to the target language concerns the GVM branch instructions. To implement tail calls correctly this must be done without stack growth. As will be explained in Section 4.2, a portable trampoline mechanism is used to jump to an arbitrary destination. We first explain the issue of data representation.

Several data types are builtin in Scheme [14, 19]: numbers, booleans, characters, strings, symbols, empty-list, pairs, vectors, promises, procedures and I/O ports. In Scheme,

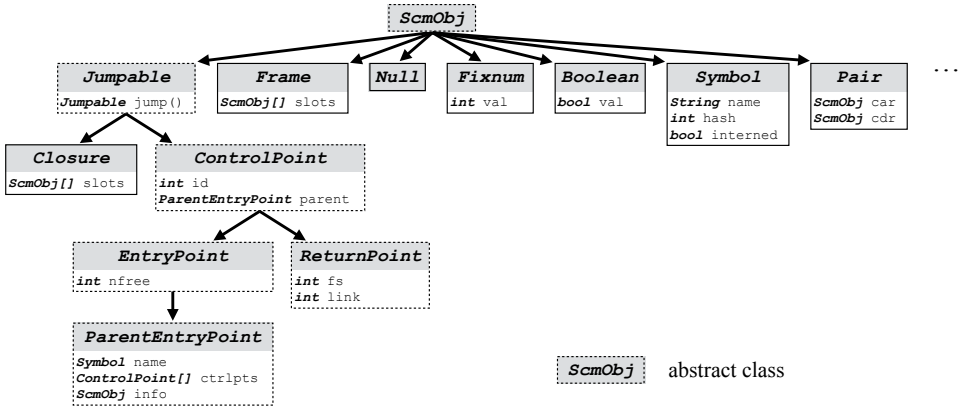

Figure 3. Class hierarchy for Scheme objects, suitable for a class-based host language such as Java.

strings are mutable and symbols are equivalent to immutable strings. Complex numbers are supported and numbers can be exact or inexact. Like other Scheme systems, Gambit implements numbers using multiple concrete types: *fixnum* for limited precision exact integers, *bignum* for unlimited precision exact integers, *ratnum* for exact rationals, *flonum* for inexact real numbers (IEEE754 floating point numbers), and *cpxnum* for complex numbers with non-zero imaginary part. Gambit also provides, as extensions to the standard, other useful types including user-defined structure types (*structures*), dictionaries (*tables*), weak references (*wills*), vectors of 8/16/32/64 bit integers and 32/64 bit floats (*homogeneous vectors*), and an "undefined value" (*void*).

For a class-based target language the data types supported by Gambit are represented using a class hierarchy defined in the Gambit runtime system. Figure 3 gives the relevant parts of the class hierarchy used by the Java back-end. The root class is `ScmObj` and most types are directly derived from this abstract class. The exception is the `Jumpable` abstract class which is used in the implementation of procedures, closures and the trampoline mechanism (Section 4.2).

With the dynamically typed target languages (JavaScript, PHP, Python and Ruby) it is advantageous for performance reasons to map the Scheme types to similar types in the target language, and only use an object-based representation when required. For example, the Scheme boolean values are mapped to the target language's boolean values, and the empty list and void values are respectively mapped to `null` and `undefined` in JavaScript, and in Python the empty list is an instance of the `Null` class of the Gambit runtime system and the void value is Python's builtin `None` value.

For all the dynamically typed target languages *fixnums* are mapped to native numbers. This has the advantage of avoiding boxing and unboxing operations when operating on small integers, a fairly common operation. All other types of numbers are boxed (instances of the class `Bignum`, `Flonum`, `Ratnum` or `Cpxnum` of the Gambit runtime system). The following types may be mapped to a native type of the target language: symbols are mapped to JavaScript strings and Ruby symbols; vectors are mapped to arrays except for PHP due to PHP's awkward array reference semantics.

Given that values of type `Jumpable` have a single `jump` method and all the dynamically typed target languages support attaching attributes to functions, `Jumpable` values are represented using functions of the target language, except for PHP where `Jumpable` and its derived classes are implemented using PHP classes (for compatibility with versions of PHP prior to 5.3).

Other types, such as pairs, strings and characters are represented using classes.

4. GVM Code Translation

This section explains the translation of the GVM code to the target language. Except for syntactic differences, similar code is generated for all target languages. For brevity we explain the translation process using JavaScript as the target language and mention cases where substantially different code is generated for another target. The `foreach` function is used as an example. Figure 4 gives the relevant parts of the JavaScript code produced.

To avoid name clashes with other code, all JavaScript global variables and function names have a distinguishing prefix (`Gambit_`). For presentation purposes, this prefix has been stripped and some minor syntactic changes have been made such as removing redundant parentheses and braces. Some optimizations discussed in Section 4.3 have also been disabled to improve readability.

4.1 GVM State

Efficient access to the GVM state is critical to achieve good execution speed. For this reason, JavaScript global variables are used (lines 1-6 in Figure 4). The stack is implemented with a JavaScript array and the global variable table is a dictionary. 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. Other dynamic target languages use a similar arrangement. In the case of Java, the runtime system defines the GVM state using static fields, and the stack is a fixed size array.

4.2 Basic CFG Translation

If we discount the branch destination inlining optimization that is explained in the next section, the back-end translates each basic block to a parameter-less JavaScript function that is conceptually an instance of `ControlPoint`. Most GVM instructions are translated straightforwardly to JavaScript code. The branch instruction at the end of the basic block is translated to a `return` of the destination operand, that is conceptually a `Jumpable` and concretely a reference to the JavaScript function containing the code of the destination basic block, or a JavaScript closure (see Section 4.6).

For example, the GVM branch instruction at the end of basic block #1 is translated at line 26 to a `return` of a reference to function `bb3_foreach`, which corresponds to basic block #3.

A trampoline, implemented by the function `trampoline` at line 8, is used to sequence the flow of control from the source to destination basic blocks. The program is started by calling `trampoline` with a reference to the basic block of the program's entry point.

The `poll` function called at lines 37 and 54 is needed for interrupt handling. After checking for interrupts, the `poll` function returns its argument if no interrupts occurred, otherwise it returns the function that handles the interrupt.

4.3 Optimizations

With the basic translation each GVM branch incurs the run time cost of one function return and call. These optimizations reduce the cost of the trampoline and interrupt checks:

Branch destination inlining. Basic blocks that 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 30.

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 26 is optimized to `return bb3_foreach();`. This allows the JavaScript VM to optimize the control flow and perhaps inline the body of the destination function. This will accumulate 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 unwinding 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 instruc-

```
1.  var r0, r1, r2, r3, r4; // GVM registers
2.  var nargs;              // argument count
3.  var stack = [null];     // runtime stack
4.  var sp = 0;             // stack pointer
5.  var glo = {};           // Scheme global vars
6.  var peps = {};          // parent entry points
7.
8.  function trampoline(pc) {
9.    while (pc !== null)
10.     pc = pc(); // call jump method
11. }
12.
13. function poll(dest) {
14.   // ...check for interrupts here...
15.   return dest;
16. }
17.
18. function Pair(car,cdr) { // pair constructor
19.   this.car = car;
20.   this.cdr = cdr;
21. }
22.
23. function bb1_foreach() { // ParentEntryPoint
24.   if (nargs !== 2)
25.     return wrong_nargs(bb1_foreach);
26.   return bb3_foreach;
27. }
28.
29. function bb3_foreach() { // Jumpable
30.   if (r2 instanceof Pair) {
31.     stack[sp+1] = r0;
32.     stack[sp+2] = r1;
33.     stack[sp+3] = r2;
34.     r1 = r2.car;
35.     r0 = bb2_foreach; // return address
36.     sp += 3;
37.     return poll(bb5_foreach);
38.   } else {
39.     r1 = false;
40.     return r0;
41.   }
42. }
43.
44. function bb5_foreach() { // Jumpable
45.   nargs = 1;
46.   return stack[sp-1]; // non-tail call f
47. }
48.
49. function bb2_foreach() { // ReturnPoint
50.   r2 = stack[sp].cdr;
51.   r1 = stack[sp-1];
52.   r0 = stack[sp-2];
53.   sp -= 3;
54.   return poll(bb3_foreach); // tail call loop
55. }
56. bb2_foreach.id = 1;
57. bb2_foreach.parent = bb1_foreach;
58. bb2_foreach.fs = 3;
59. bb2_foreach.link = 1;
60.
61. bb1_foreach.id = 0;
62. bb1_foreach.parent = bb1_foreach;
63. bb1_foreach.nfree = -1; // not a closure
64. bb1_foreach.name = "foreach";
65. bb1_foreach.ctrlpts = [bb1_foreach,bb2_foreach];
66. bb1_foreach.info = false; // no debug info
67.
68. peps["foreach"] = bb1_foreach;
```

Figure 4. JavaScript code generated for the `foreach` function

tion 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 37, which is a polling branch to basic block #5, is optimized to:

```
if (--pollcount === 0)
  return poll(bb5_foreach);
else { // inlined call to bb5_foreach
  nargs = 1;
  return stack[sp-1]; // non-tail call f
}
```

4.4 Stack Space Leak

The explicit management of the `stack` array raises a space leak issue. When `sp` is lowered, as in line 53, the slots beyond `sp` become garbage conceptually, but the JavaScript VM's garbage collector is not aware of this and will consider that all the values contained in those slots are live. The length of the `stack` array must be adjusted using `sp` to avoid the space leak. This reclaims the unused stack space and also prevents the garbage collector from retaining the objects in those slots that would otherwise be reachable.

The resizing of the stack could be done whenever `sp` is lowered, but this would be very expensive due to the frequent changes to `sp`. Instead, the resizing is performed by the `poll` function. This means that there is always some amount of garbage at the end of the `stack` array when the VM's garbage collection occurs. However, in a bounded time (the next call to `poll`) such garbage will truly be unreachable. The `poll` function has the following outline:

```
function poll(dest) {
  pollcount = 100;
  stack.length = sp + 1;
  // ...check for interrupts here...
  return dest;
}
```

Space leaks are also possible when dead GVM registers contain object references. The `poll` function could overwrite dead registers based on liveness information computed by the Gambit compiler, but that is not currently implemented.

4.5 Marshalling Control Points

The code generated also stores some meta information on the first-class basic blocks (functions `bb1_foreach` and `bb2_foreach`, which are conceptually instances of `ParentEntryPoint` and `ReturnPoint` respectively). This meta information is used for the implementation of first-class continuations and the marshalling and unmarshalling of procedures, closures and continuations.

To support dynamic loading of modules, basic blocks produced by the compiler are partitioned into named parent procedures. Each named top level Scheme procedure, such as the `foreach` procedure, is a parent procedure (note that a module is treated like a top level Scheme procedure that executes the initializations it contains). Nested in each named parent procedure is its proper basic blocks and the basic blocks of all its subprocedures. A unique zero based index is assigned to each first-class basic block within a parent. It is thus possible to identify a given control point by its index and parent procedure (the `id` and `parent` properties of `ControlPoint`). It is also possible to identify a given parent procedure by its name (the `name` property of `ParentEntryPoint`). This allows for the marshalling of control points.

For unmarshalling control points, the `peps` dictionary is used (see lines 6 and 68). It maps each parent procedure name to its corresponding `ParentEntryPoint`. In the `ParentEntryPoint` the property `ctrlpts` is an array of the control points contained in the parent procedure. By indexing this array with the zero based index of the control point, the JavaScript function corresponding to the control point can be recovered.

So when control point `cp` is marshalled, the values i=`cp.id` (an integer) and n=`cp.parent.name` (a symbol) are sent to the stream. When the unmarshalling node extracts these values from the stream, it recovers the control point by evaluating `peps[`n`].ctrlpts[`i`]` . Note that this requires that all the nodes run compiled programs built with the same code base, or run interpreted programs (`eval` is a compiled Scheme procedure that is in the Gambit runtime system and thus part of all programs).

For the return-point basic block #2 the properties `fs` and `link` are set at lines 58 and 59. This is required for the implementation of continuations (see Section 5).

4.6 Closures

The mapping from Scheme closures to JavaScript closures is designed to support closure marshalling. The GVM's flat closures are composed of a number of slots, one referring to the closure entry point and the rest are the Scheme closure's free variables. The JavaScript closure has two free variables: the slots of the Scheme closure (a JavaScript object) and a reference to the JavaScript closure itself.

Consider the `ccons` function (curried `cons`) whose definition is given in Figure 5 and whose generated JavaScript code is in Figure 6.

The construction of a Scheme closure is a two step process. First, it is allocated using the `closure_alloc` function (line 1). The slots of the closure are the only parameter of `closure_alloc`. The actual JavaScript closure is the `self` function defined at line 3. Normally, `self` is called with no argument (i.e. `msg` will be `undefined`). The slots of the closure are obtained by calling `self` with a single `true` argument.

The first slot is set to the closure's entry point (line 15). When the closure is called, with no argument, the first slot of the closure is accessed (line 6) to branch to the correct closure entry point. `r4` will have been set to a reference to the closure itself (line 5), so that access to free variables is possible. For example the access to `x` is translated to reading the second slot (line 22).

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

Figure 5. Source code of the `ccons` procedure

Marshalling closures consists in marshalling all of its slots. The control point of the closure's entry point (the first slot) is marshalled as explained in Section 4.5.

5. Implementing Continuations

We use the *incremental stack/heap strategy* for managing continuations [5]. This strategy allows the GVM code to use a standard procedure call protocol.

In the incremental stack/heap strategy, the current 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 58-59 in Figure 4).

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 `null` marks the end of the chain). In other words, the heap frames are chained using the slot of the frame that 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 procedure that 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 that contains the return address is set to the function `underflow`. The definition of the `underflow` function is given in Figure 8.

```
1.  function closure_alloc(slots) {
2.
3.    function self(msg) {
4.      if (msg === true) return slots;
5.      r4 = self;
6.      return slots[0];
7.    }
8.
9.    return self;
10. }
11.
12. function bb1_ccons() { // ParentEntryPoint
13.   if (nargs !== 1)
14.     return wrong_nargs(bb1_ccons);
15.   r1 = closure_alloc([bb2_ccons,r1]);
16.   return r0;
17. }
18.
19. function bb2_ccons() { // EntryPoint
20.   if (nargs !== 1)
21.     return wrong_nargs(r4);
22.   r4 = r4(true)[1];
23.   r1 = new Pair(r4,r1);
24.   return r0;
25. }
26.
27. bb2_ccons.id = 1;
28. bb2_ccons.parent = bb1_ccons;
29. bb2_ccons.nfree = 1; // 1 free var
30.
31. bb1_ccons.id = 0;
32. bb1_ccons.parent = bb1_ccons;
33. bb1_ccons.nfree = -1; // not a closure
34. bb1_ccons.name = "ccons";
35. bb1_ccons.ctrlpts = [bb1_ccons,bb2_ccons];
36. bb1_ccons.info = false;
37.
38. peps["ccons"] = bb1_ccons;
```

Figure 6. JavaScript code generated for `ccons`

The `heapify_cont` function given in Figure 7 implements continuation heapification for JavaScript. The parameter `ra` is the return address back to the procedure that created the topmost continuation frame. The algorithm iterates over the stack frames from top to bottom to create a heap copy. The oldest stack frame is not copied. Instead, the stack array is simply reused after shrinking it to the right size. At the end of heapification, the raw representation of the current continuation is in `stack[0]`.

With the `heapify_cont` function, it is easy to implement the continuation API of [7]. The Scheme procedures

```
1.  function heapify_cont(ra) {
2.
3.    if (sp > 0) { // stack has >= 1 frame
4.
5.      var fs = ra.fs, link = ra.link;
6.      var base = sp - fs;
7.      var chain;
8.
9.      if (base > 0) { // stack has >= 2 frames
10.       chain = stack.slice(base,base+fs+1);
11.       chain[0] = ra;
12.       sp = base;
13.       var prev_frm = chain, prev_link = link;
14.       ra = prev_frm[prev_link];
15.       fs = ra.fs;
16.       link = ra.link;
17.       base = sp - fs;
18.
19.       while (base > 0) {
20.         var frm = stack.slice(base,base+fs+1);
21.         frm[0] = ra;
22.         sp = base;
23.         prev_frm[prev_link] = frm;
24.         prev_frm = frm; prev_link = link;
25.         ra = prev_frm[prev_link];
26.         fs = ra.fs;
27.         link = ra.link;
28.         base = sp - fs;
29.       }
30.
31.       stack[link] = stack[0];
32.       stack[0] = ra;
33.       stack.length = fs + 1;
34.       prev_frm[prev_link] = stack;
35.
36.     } else {
37.       stack[link] = stack[0];
38.       stack[0] = ra;
39.       stack.length = fs + 1;
40.       chain = stack;
41.     }
42.
43.     stack = [chain];
44.     sp = 0;
45.   }
46.
47.   return underflow;
48. }
```

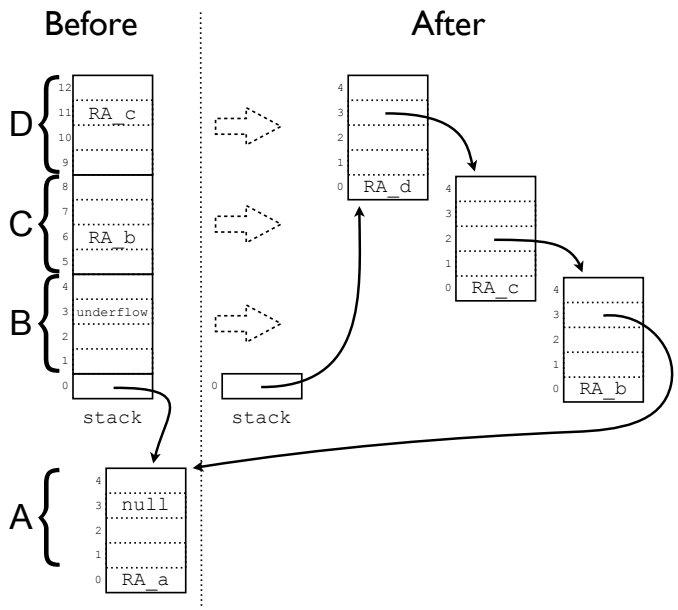

Figure 7. Continuation heapification algorithm and example. Before heapification, continuation frames B, C and D are on the stack. After heapification with the call `heapify_cont(RA_d)`, where `RA_d` is the return address back to the procedure that created frame D, all frames are in the heap and explicitly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

`continuation-capture` and `continuation-return` are the primitive continuation manipulation operations. These procedures are implemented by the JavaScript code given in Figure 9. `continuation-capture` is similar to `call/cc` but the continuation passed to the receiver procedure is a raw continuation (not wrapped in a closure). It is implemented by calling the `heapify_cont` function with the current return address and then passing `stack[0]` to the receiver procedure. `continuation-return` takes a raw continuation and a value, and resumes the continuation with that value as its result. It is implemented by returning to the `underflow` function after setting up an empty stack with `stack[0]` referring to the continuation to resume.

`call/cc` is a thin wrapper over these primitives. The raw continuation produced by `continuation-capture` is wrapped in a closure, which is what `call/cc`'s receiver expects. The implementation is given in Figure 10.

Once heapified, a continuations can be marshalled easily because each frame is essentially a closure (the return address is a control point similar to a closure's entry point).

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

Figure 8. Definition of the `underflow` function

```
1.  function bb1_continuation_capture() {
2.    var receiver = r1;
3.    r0 = heapify_cont(r0);
4.    r1 = stack[0];
5.    nargs = 1;
6.    return receiver;
7.  }
8.
9.  function bb1_continuation_return() {
10.   sp = 0;
11.   stack[0] = r1;
12.   r0 = underflow;
13.   r1 = r2;
14.   return r0;
15. }
```

Figure 9. Implementation of the continuation primitives `continuation-capture` and `continuation-return`

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

Figure 10. Definition of `call/cc`

6. Evaluation

Our main interest is evaluating the execution speed of our approach, that is, the speed for executing various styles of programs. The compilation time is of little concern as there is no reason to believe that it would vary significantly between competing approaches.

Evaluating execution speed can be done by comparing it to existing compilers targeting dynamic languages. The only mature Scheme compilers we know of that target dynamic languages are those targeting JavaScript, specifically Scheme2JS and Spock. For that reason we limit our comparison to those systems and only use the Gambit JavaScript back-end (which we call Gambit-JS). Moreover, it is interesting to do this evaluation using several state-of-the-art JavaScript VMs to see how this affects execution speed.

6.1 Methodology

Our methodology consists in executing with each system specially selected benchmark programs that represent use-cases of non-tail calls, tail calls and first-class continuations.

Although it has the virtue of being empirical, the methodology has pitfalls for comparing the implementation 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, giving one compiler an advantage that is not related to the continuation implementation approach. We are interested here in comparing the approaches as implemented by the compilers, not the compilers as a whole. 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 examined 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 parts of the source program 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). For example, a variant of the `fib35` benchmark where the recursive function is local to another function is compiled by Scheme2JS to JavaScript code that runs 7 times faster on V8 than when the recursive function is global. 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, programs using non-primitive library functions have been avoided or they contain a generic Scheme definition of the function with calls to 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 (i.e. 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. Gambit-JS and Scheme2JS do user-function inlining differently and under different condi-

tions. 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 the `fib35` benchmark that the overhead is less than 5%).

Note that Scheme2JS and Spock do not support marshalling of closures and continuations, which is the motivation for our work.

6.2 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 programs in the second group use `call/cc` in various ways. The programs `ctak` and `contfib30` have non-tail-recursive functions of moderate recursion depth: `ctak` reifies each continuation of its recursion, and `contfib30` reifies only the continuations at the leaves of the recursion. The remaining programs have a shallow call graph (i.e. the current continuation is only a few frames deep when `call/cc` is called). The program `btsearch2000` performs a backtracking search, and `threads10` is a thread scheduler that interleaves the execution of 10 threads.

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

6.3 Setting

A computer with a 2.6 GHz Intel Core i7 processor and 16 GB RAM is used in all the experiments. The latest versions of four popular web browsers are used: Microsoft Edge 20.10240.16384.0, Google Chrome 44.0.2403.157, Mozilla Firefox 40.0.2, and Apple Safari 8.0.6 (Chakra, V8, SpiderMonkey, and Nitro JavaScript VMs respectively). Microsoft Edge is run on Windows 10.0.10240, and all other browsers are run on OS X 10.10.5. We use more than one JavaScript VM in order to evaluate the effect of the choice of VM. The performance of a system obviously depends on how well the VM optimizes the style of JavaScript code generated by the Scheme system (code structure, amenability to analysis, set of language constructs used, etc). As the style departs from what is expected of "normal" code, there is a higher likelihood that the authors of the VM have not invested the effort to implement an optimization for that style of code.

The Scheme systems used are:

- Gambit-JS version 4.7.8 with the declarations `(declare (standard-bindings) (fixnum) (not safe) (block) (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.

6.4 Results

The execution times of the benchmark programs using Chakra, V8, SpiderMonkey, and Nitro are given in Table 1 a-d respectively. The times in seconds is given (average of 20 runs), and for Scheme2JS and Spock, the ratio with the Gambit-JS time is also given.

Gambit-JS is consistently faster than the other systems on all JavaScript VMs tested and its best absolute execution times are obtained using Chakra.

If we focus on Table 1a, which gives the times on Chakra, we see that Gambit-JS is 3.2 to 490.7 times faster than Scheme2JS, and 9.4 to 104.9 times faster than Spock. For reference, Gambit-JS on Chakra is on average 7 times slower than when using the Gambit-C compiler (compilation to C then to native code).

Scheme2JS has its best relative times when `call/cc` is not used (3.2 to 29.2 times slower). When `call/cc` is used, the performance depends greatly on the depth of the continuation where the `call/cc` is called (82.6 to 490.7 times slower). The largest slowdowns are for `ctak` and `contfib30`. These programs call `call/cc` in moderately deep recursions and there is repetitive capturing of (parts of) the continuations. The large slowdown is explained by the fact that the Replay-C algorithm copies and restores the complete continuation on the JavaScript VM stack every time a continuation is captured and invoked. Our approach only copies the frames that have not yet been captured and restores continuations incrementally, one frame at a time, so the cost does not depend on the depth of the continuation. When using Nitro, Table 1d, the slowdown increases dramatically to 709.1× on `contfib30`. This is probably due to a higher hidden constant on that VM for copying/restoring the stack, which is related to the cost of throwing and catching exceptions to iterate over the stack frames.

The CPS conversion used by Spock makes it straightforward to reify continuations because all functions are passed an explicit continuation parameter. Unsurprisingly, Spock has its best relative times when `call/cc` is used. If we focus on Table 1d (Nitro), we see that Nitro is 56.4 to 123.1

Program	Gambit-JS	Scheme2JS		Spock	
`fib35`	.39	11.30	29.2×	14.32	37.0×
`nqueens12`	.49	2.92	5.9×	11.90	24.2×
`oddeven`	.39	1.23	3.2×	40.79	104.9×
`ctak`	.12	57.90	490.7×	1.62	13.7×
`contfib30`	.94	353.31	375.9×	8.82	9.4×
`btsearch2000`	.94	77.78	82.6×	14.15	15.0×
`threads10`	.64	71.27	112.2×	16.99	26.7×

a. Execution times using Chakra (Microsoft Edge 20.10240.16384.0)

Program	Gambit-JS	Scheme2JS		Spock	
`fib35`	.52	7.91	15.3×	31.47	61.0×
`nqueens12`	.64	3.97	6.2×	17.01	26.8×
`oddeven`	.43	1.18	2.8×	51.37	119.7×
`ctak`	.15	21.68	140.8×	2.33	15.1×
`contfib30`	1.08	119.70	111.0×	13.21	12.3×
`btsearch2000`	1.20	22.39	18.7×	19.07	16.0×
`threads10`	.85	28.16	33.0×	28.67	33.7×

b. Execution times using V8 (Google Chrome 44.0.2403.157)

Program	Gambit-JS	Scheme2JS		Spock	
`fib35`	.52	1.26	2.4×	4.39	8.4×
`nqueens12`	.70	.77	1.1×	3.69	5.2×
`oddeven`	.41	1.01	2.5×	9.50	23.3×
`ctak`	.14	30.90	227.2×	.68	5.0×
`contfib30`	.80	155.12	192.9×	3.66	4.6×
`btsearch2000`	1.16	37.71	32.5×	8.05	6.9×
`threads10`	.78	33.98	43.7×	6.48	8.3×

c. Execution times using SpiderMonkey (Mozilla Firefox 40.0.2)

Program	Gambit-JS	Scheme2JS		Spock	
`fib35`	.37	1.81	4.9×	45.06	122.1×
`nqueens12`	.42	.66	1.6×	23.46	56.4×
`oddeven`	.61	.76	1.3×	74.85	123.1×
`ctak`	.14	90.47	650.9×	2.38	17.1×
`contfib30`	.86	609.79	709.1×	14.39	16.7×
`btsearch2000`	1.80	73.73	41.0×	24.92	13.9×
`threads10`	.76	72.06	94.2×	40.42	52.8×

d. Execution times using Nitro (Apple Safari 8.0.6)

Table 1. Execution times in seconds using various JavaScript VMs

times slower when `call/cc` is not used but only 13.9 to 52.8 times slower when `call/cc` is used. We suspect that when `call/cc` is not used the creation of closures for the continuation frames of non-tail calls is more expensive than using an explicit representation on a stack as in Gambit-JS. It is surprising that for `oddeven`, which only performs tail calls (i.e. no continuation frames are created), the relative time goes up to 123.1×. 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 on branch instructions, but not intermittently. When a counter is manually added to check intermittently, the time is roughly halved, which is still much 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 Nitro optimizer (in particular the Spock stack checks use the JavaScript `arguments` form, which is known to disable some optimizations).

Program	Gambit-JS			Scheme2JS			Spock		
	V8	SM	N	V8	SM	N	V8	SM	N
`fib35`	1.3	1.3	1.0	.7	.1	.2	2.2	.3	3.1
`nqueens12`	1.3	1.4	.8	1.4	.3	.2	1.4	.3	2.0
`oddeven`	1.1	1.0	1.6	1.0	.8	.6	1.3	.2	1.8
`ctak`	1.3	1.2	1.2	.4	.5	1.6	1.4	.4	1.5
`contfib30`	1.1	.9	.9	.3	.4	1.7	1.5	.4	1.6
`btsearch2000`	1.3	1.2	1.9	.3	.5	.9	1.3	.6	1.8
`threads10`	1.3	1.2	1.2	.4	.5	1.0	1.7	.4	2.4
geo. mean	*1.2*	*1.2*	*1.2*	*.5*	*.4*	*.7*	*1.5*	*.4*	*2.0*

SM = SpiderMonkey, N = Nitro

Table 2. Relative execution times for each benchmark and Scheme system using Chakra, V8, SpiderMonkey, and Nitro. The baseline is the time for that benchmark on that Scheme system using Chakra.

We will now examine how the performance of the Scheme systems varies across JavaScript VMs. This is an important issue because a web developer has no control over the web browser used by the clients. The code must run reasonably fast on all the popular JavaScript VMs. Table 2 gives the relative execution times for each benchmark and Scheme system using V8, SpiderMonkey, and Nitro, relative to Chakra.

For Gambit-JS on V8, the benchmarks run 1.1 to 1.3 times slower than they do on Chakra. On SpiderMonkey, the benchmarks run 0.9 to 1.4 times slower than they do on Chakra. On Nitro, the benchmarks run 0.8 to 1.9 times slower than they do on Chakra. For each of these VMs, the geometric mean of the slowdowns is 1.2×. The compactness of these ranges and the consistent average slowdown suggests that for Gambit-JS the performance of a program varies little between VMs. In fact, across all the benchmarks and VMs, the maximum execution speed ratio is 1.9× (`btsearch2000` on Nitro and Chakra).

Consequently, for Gambit-JS, it is feasible to use the performance of a program on Chakra to predict its approximate performance on V8, SpiderMonkey, and Nitro. Performance is much less predictable for the other systems. Scheme2JS has its best execution times on SpiderMonkey where it is 2.6× faster than Chakra on average and up to 9× faster (`fib35`). Similarly Spock has its best execution times on SpiderMonkey where it is 2.8× faster than Chakra on average and up to 4.3× faster (`oddeven`).

Of course, Gambit-JS is not completely immune to the set of optimizations performed by the JavaScript VM. In fact performance anomalies were encountered in tests of Gambit-JS on earlier versions of Firefox (version 15.0.1). With that version of Firefox the range of the slowdowns for Gambit-JS is about the same as for Firefox 40.0.2 for the benchmarks in the first group, and roughly 4 times larger than for Firefox 40.0.2 for the benchmarks in the second group. One might think that the cause of the higher slowdowns in the second group is the use of `call/cc`, but further investigation reveals that the issue is the implementation of closures in the Gambit-JS back-end. All programs in the second

group create and call closures frequently, and especially so in the `btsearch2000` benchmark, which has the highest slowdown. Our implementation of closures appears to be handled by that version of SpiderMonkey less efficiently than by Chakra, V8, and Nitro. This has been verified with a test program that creates and calls closures in a tight loop. We can imagine that older versions of the tested JavaScript VMs or current VMs we haven't tested and that implement less effective optimizations would also have less predictable performance. However, we believe that the simple constructs, in particular the avoidance of `try/catch` and `arguments`, and the simple code structure used in the output of Gambit-JS make it more likely that a JavaScript VM will compile the code efficiently.

7. Conclusion

We have proposed a VM-based approach for implementing tail calls without stack growth and first-class continuations in a Scheme compiler targeting Java, JavaScript, PHP, Python and Ruby. The approach supports marshalling and unmarshalling closures, continuations and other Scheme data, allowing the migration of computational tasks between the supported target languages.

The compiler uses an intermediate level representation, the Gambit Virtual Machine (GVM), which is 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. We use the incremental stack/heap strategy [5], which allows the GVM code to use a standard function call protocol with zero overhead for code that doesn't manipulate first-class continuations and a cost for invoking a continuation that is proportional to the size of the topmost continuation frame.

Our experiments on specially selected benchmark programs on four popular JavaScript VMs show that the approach compares favorably to the Replay-C algorithm used in the Scheme2JS compiler and to the Cheney on the MTA approach used in the Spock compiler. The execution time is consistently faster for our approach. When comparing Gambit-JS to Scheme2JS, Gambit-JS is 3.2 to 491 times faster on Chakra, 2.8 to 141 times faster on V8, 1.1 to 227 times faster on SpiderMonkey, and 1.3 to 709 times faster on Nitro. When comparing Gambit-JS to Spock, Gambit-JS is 9.4 to 105 times faster on Chakra, 12.3 to 120 times faster on V8, 4.6 to 23 times faster on SpiderMonkey, and 14 to 123 times faster on Nitro. Our experiments also show that the performance of the code generated by Gambit-JS is more predictable across VMs than for the other systems.

Acknowledgments

We wish to thank Florian Loitsch, Manuel Serrano and Felix Winkelmann for helping us understand their systems and Eric Thivierge and Alexandre St-Louis Fortier for assisting with preliminary work. This work was supported by the Natural Sciences and Engineering Research Council of Canada and Mozilla Corporation.

References

[1] J. Ashkenas. List of languages compiling to JavaScript, 2015. URL `https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS/`.

[2] H. G. Baker. Cons should not cons its arguments, part II: Cheney on the M.T.A. *SIGPLAN Notices*, 30(9):17–20, 1995.

[3] E. Brady. Cross-platform compilers for functional languages. In *Proceedings of the 2015 Trends in Functional Programming Symposium*, June 2015.

[4] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Trans. Program. Lang. Syst.*, 17(5): 704–739, Sept. 1995. ISSN 0164-0925.

[5] W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher Order and Symbolic Computation*, 12(1):7–45, 1999.

[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.

[7] M. Feeley. A better API for first-class continuations. In *Scheme and Functional Programming Workshop*, SFPW '01, pages 1–3, 2001.

[8] M. Feeley. Gambit-C version 4.7.8, 2015. URL `http://gambitscheme.org`.

[9] 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.

[10] G. Germain, M. Feeley, and S. Monnier. Concurrency oriented programming in Termite Scheme. In *Scheme and Functional Programming Workshop*, SFPW'06, pages 125–135, 2006.

[11] Haxe Foundation. Haxe 3 Manual, Apr. 2015. URL `http://haxe.org`.

[12] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, volume 25, pages 66–77, New York, NY, USA, 1990.

[13] S. Jagannathan. Continuation-based transformations for coordination languages. *Theoretical Computer Science*, 240(1): 117 – 146, 2000. ISSN 0304-3975.

[14] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

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

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

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

[18] 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, 1995. Springer-Verlag.

[19] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19:1–301, 2009. ISSN 1469-7653.

[20] F. Winkelmann. Chicken-Spock, 2014. URL `http://wiki.call-cc.org/eggref/4/spock`.

[21] F. Winkelmann. Chicken Scheme, 2015. URL `http://www.call-cc.org/`.

[22] D. Yoo. *Building Web Based Programming Environments for Functional Programming*. PhD thesis, Worcester Polytechnic Institute, 2012.

A. Source Code of Benchmark Programs

```
(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-bench "nqueens12" (lambda () (nqueens 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-bench "oddeven" (lambda () (odd 100000000)))
```

Source code of `oddeven`

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))

(run-bench "fib35" (lambda () (fib 35)))
```

Source code of `fib35`

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

      (call/cc
       (lambda (k)
         (k 1)))

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

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

Source code of `contfib30`

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

(define (in-range a b)
  (call/cc
   (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-bench
 "btsearch2000"
 (lambda () (btsearch 2000)))
```

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/cc
    (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/cc
   (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-bench
 "threads10"
 (lambda () (threads 10)))
```

Source code of `threads10`

```
(define (ctak x y z)
  (call/cc
   (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/cc
        (lambda (k) (ctak-aux k (- x 1) y z)))
       (call/cc
        (lambda (k) (ctak-aux k (- y 1) z x)))
       (call/cc
        (lambda (k) (ctak-aux k (- z 1) x y))))))

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

Source code of `ctak`