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 JavaScript, a dynamic language without those features. Our approach is based on the use of a simple custom virtual machine intermediate representation that is translated to JavaScript. 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 systems with a set of benchmark programs on three popular JavaScript VMs (V8, JägerMonkey and Nitro). On the benchmark programs, all systems perform best when executed with V8 and our approach is consistently faster than the others on all VMs. For some VMs and benchmarks our approach is moderately faster than the others (below a factor of 2), but in some cases there is a very large performance gap (with Nitro there is a slowdown of up to 3 orders of magnitude for Scheme2JS, and up to 2 orders of magnitude for Spock).

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 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 languages are C, C++, Java, Python, Haskell, and Scheme.

Using a popular dynamic language such as JavaScript, Python or Ruby as the target language for a Scheme compiler is alluring because they offer dynamic typing, introspective features, closures and garbage collection that 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, in the general case, have no direct equivalent 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 can also execute in web browsers.

A Scheme system 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 targetting high-level languages such as the Bigloo [14], Chicken [1] and Gambit-C [7] Scheme to C compilers, and the Scheme2JS [11–13], 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. By using a CPS conversion of the code, the Cheney on the MTA approach makes it possible to reclaim all stack frames. 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 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 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SFP '12 9 September 2012, Copenhagen.
Copyright © 2012 ACM . . . $10.00

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 [10]. In this paper we use a similar virtual machine based compilation approach but targeting JavaScript.

These approaches offer different tradeoffs. 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.

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 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 garbage collecting 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 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. This also causes overhead because optimization by the target VM is hindered.

In this paper we demonstrate that an approach for implementing tail calls and first-class continuations 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 that is consistently faster than both Scheme2JS and Spock on the three JavaScript VMs we have tested, and on some JavaScript VMs the performance gap is very large.

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 gives 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 which 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 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 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. 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

```
1.  (declare (standard-bindings)
             (not safe)
             (inlining-limit 0))

5.  (define (foreach f lst)
      (let loop ((lst lst))
        (if (pair? lst)
            (begin
              (f (car lst))
              (loop (cdr lst)))
            #f)))
```

Figure 1. Source code of the `foreach` function

```
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

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_{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 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 which guarantee a bounded number of GVM instructions executed between interrupt checks.

3. Translation to JavaScript

The main difficulty in translating GVM code to JavaScript concerns the GVM branch instructions which transfer control from a source to destination location. To implement tail calls correctly this must be done without stack growth. We will give the details of the trampoline approach used by Gambit-JS starting at Section 3.3. We first need to address some ancillary issues.

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.

```
1.  var r0, r1, r2, r3, r4; // 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 trampoline(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 (r2 instanceof Pair) {
26.     stack[sp+1] = r0;
27.     stack[sp+2] = r1;
28.     stack[sp+3] = r2;
29.     r1 = r2.car;
30.     r0 = bb2_foreach;
31.     sp += 3;
32.     return poll(bb5_foreach);
33.   } else {
34.     r1 = false;
35.     return r0;
36.   }
37. }
38.
39. function bb2_foreach() { // return-point
40.   r2 = stack[sp].cdr;
41.   r1 = stack[sp-1];
42.   r0 = 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

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.

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` of 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 3.7).

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 `trampoline` at line 12, 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 32 and 44 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.

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 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 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 Stack Space Leak

The explicit management of the `stack` array raises a space leak issue. When `sp` is lowered, 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 a 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) {
  poll_count = 100;
  stack.length = sp + 1;
  ... check for interrupts ...
  return dest;
}
```

3.6 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.

3.7 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 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 4 and whose generated JavaScript code is in Figure 5.

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 `false` argument.

The property `v0` of the slots is set to the closure's entry-point (line 16). When the closure is called, with no argument, the property `v0` of the closure's slots 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 closed variables is possible. For example the access to `x` is translated to reading property `v1` of the slots (line 26).

4. Implementing Continuations

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

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

Figure 4. Source code of the `ccons` function

```
1.  function closure_alloc(slots) {
2.
3.    function self(msg) {
4.      if (msg === false) return slots;
5.      r4 = self;
6.      return slots.v0;
7.    }
8.
9.    return self;
10. }
11.
12. function bb1_ccons() { // entry-point
13.   if (nargs !== 1)
14.     return wrong_nargs(bb1_ccons);
15.   var closure1 =
16.     closure_alloc({v0:bb2_ccons,v1:r1});
17.   stack[sp+1] = closure1;
18.   r1 = stack[sp+1];
19.   return r0;
20. }
21. bb1_ccons.id = "bb1_ccons"; // meta info
22.
23. function bb2_ccons() { // closure-entry-point
24.   if (nargs !== 1)
25.     return wrong_nargs(bb2_ccons);
26.   r4 = r4(false).v1;
27.   r1 = new Pair(r4, r1);
28.   return r0;
29. }
30. bb2_ccons.id = "bb2_ccons"; // meta info
```

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

```
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

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

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

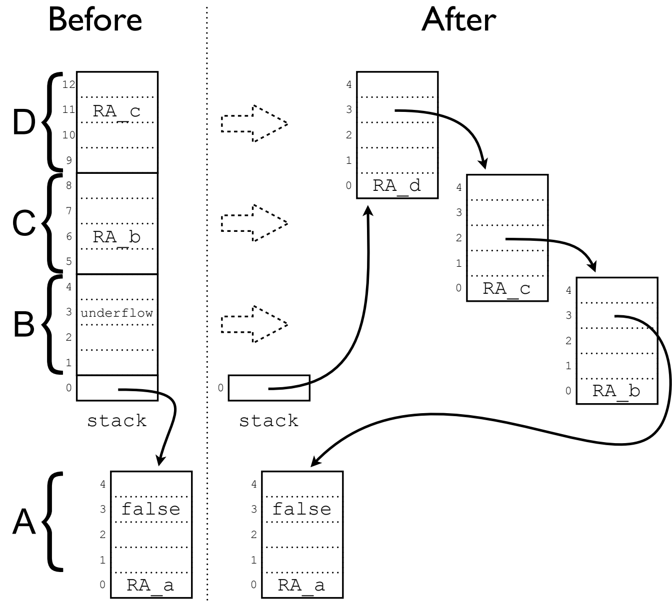

Figure 7. Continuation heapification algorithm and example. Before heapification, continuation frames B, C and D are on the stack. After heapification with the call `heapify(RA_d)`, where `RA_d` is the return address back to the function that created frame D, all frames are in the heap and explicitly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

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 3).

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` 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 function 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 6.

Continuation heapification is implemented with the `heapify` function given in Figure 7. The parameter `ra` is the return address back to the function 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` function, it is easy to implement the continuation API of [7]. The Scheme functions `continuation-capture` and `continuation-return` are the primitive continuation manipulation functions. These functions are implemented by the JavaScript code given in Figure 8.

```
1.  function bb1_continuation_capture() {
2.    var receiver = r1;
3.    r0 = heapify(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 8. Implementation of the continuation primitives `continuation-capture` and `continuation-return`

The function `continuation-capture` is similar to `call/cc` but the continuation passed to the receiver function is a raw continuation (not wrapped in a closure). It is implemented by calling the `heapify` function with the current return address and then passing `stack[0]` to the receiver function. The function `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]` refering to the continuation to resume.

The `call/cc` function is a thin wrapper over these primitives. It wraps the raw continuation produced by `continuation-capture` in a closure, which is what `call/cc`'s receiver expects. The implementation is simply:

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

5. Evaluation

In this section we aim to evaluate the performance of the three approaches discussed in this paper and that are implemented in the Gambit-JS, Scheme2JS and Spock compilers. We are interested in evaluating the execution speed. We are not concerned with compilation time (there is no reason to believe that the compilation time of the different approaches would be significantly different and in any case most of the compilation time is expected to be in writing out the JavaScript code).

5.1 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 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, giving 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 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. 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 the `fib35` benchmark 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 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.

Program	Gambit-JS	Scheme2JS		Spock	
fib35	.80	1.54	1.9×	2.40	3.0×
nqueens12	.72	.76	1.1×	2.33	3.3×
oddeven	.83	1.92	2.3×	5.62	6.8×
ctak	.18	17.64	95.9×	.59	3.2×
contfib30	1.17	106.01	90.9×	3.38	2.9×
btsearch2000	1.35	25.40	18.8×	7.93	5.9×
threads10	1.34	24.68	18.5×	4.40	3.3×

Table 1. Execution times using V8 (Chrome 21.0.1180.89)

Program	Gambit-JS	Scheme2JS		Spock	
fib35	1.07	7.66	7.2×	20.73	19.4×
nqueens12	1.49	2.62	1.8×	12.97	8.7×
oddeven	1.09	1.93	1.8×	33.91	31.1×
ctak	1.25	30.38	24.3×	1.86	1.5×
contfib30	5.88	156.79	26.7×	10.61	1.8×
btsearch2000	11.19	27.16	2.4×	16.54	1.5×
threads10	6.97	34.36	4.9×	22.11	3.2×

Table 2. Execution times using JägerMonkey (Firefox 15.0.1)

Program	Gambit-JS	Scheme2JS		Spock	
fib35	1.32	6.53	4.9×	84.69	64.2×
nqueens12	1.31	1.89	1.4×	45.15	34.4×
oddeven	2.52	1.55	.6×	143.97	57.1×
ctak	.30	52.88	177.4×	4.18	14.0×
contfib30	1.60	2311.69	1443.0×	25.98	16.2×
btsearch2000	3.46	102.98	29.7×	92.02	26.6×
threads10	3.03	74.21	24.5×	74.88	24.7×

Table 3. Execution times using Nitro (Safari 6.0)

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

5.3 Setting

An OS X 10.8.1 computer with a 2.2 GHz Intel Core i7 processor and 16 GB RAM is used in all the experiments. The latest versions of three popular JavaScript VMs are used to see how performance varies between VMs. Chrome 21.0.1180.89, Firefox 15.0.1, and Safari 6.0 are used (V8, JägerMonkey and Nitro JavaScript VMs respectively). The Scheme systems used are:

- Gambit-JS version v4.6.6 20120908010706 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 using V8, JägerMonkey and Nitro are given in Tables 1, 2 and 3. The times in seconds is given, and for Scheme2JS and Spock, the ratio with the Gambit-JS time is also given.

For each JavaScript VM, Gambit-JS is consistently faster than the other systems. The only anomaly occurs on Nitro where Scheme2JS runs `oddeven` faster than Gambit-JS.

For each Scheme system, the best times are obtained when using V8. The only anomaly is that Scheme2JS runs the `oddeven` on Nitro slightly faster than on V8.

If we focus on Table 1, which gives the times on V8, we see that Gambit-JS is 1.1 to 95.9 times faster than Scheme2JS, and 2.9 to 6.8 times faster than Spock.

Program	JägerMonkey	Nitro
fib35	1.3×	1.7×
nqueens12	2.1×	1.8×
oddeven	1.3×	3.0×
ctak	6.8×	1.6×
contfib30	5.0×	1.4×
btsearch2000	8.3×	2.6×
threads10	5.2×	2.3×

Table 4. Effect of different JavaScript VMs on execution times (baseline is V8)

Scheme2JS has its best relative times when `call/cc` is not used (1.1 to 2.3 times slower). When `call/cc` is used, the performance depends greatly on the depth of the continuation where the `call/cc` is called (18.5 to 95.9 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, the slowdown increases dramatically to 1443 times slower on `contfib30`. This is probably due to a higher hidden constant on that VM for copying/restoring the stack (such as the cost of throwing and catching exceptions to iterate over the stack frames).

The CPS conversion used by Spock makes it trivial to reify continuations because all functions are passed an explicit continuation parameter. Unsurprisingly, Spock has its best relative times when `call/cc` is used. This is most apparent when using JägerMonkey and Nitro. If we focus on Table 1 (V8), we see that when `call/cc` is not used the relative times for Spock range from 3 to 3.3 times slower 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 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 6.8. 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 added to check intermittently, the time is roughly halved, which is still 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 is known to disable some optimizations).

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.

For Gambit-JS, the relative execution times of the benchmarks on JägerMonkey and Nitro (using V8 as a baseline) are given in Table 4. The benchmarks run 1.4 to 3 times slower on Nitro than they do on V8. The compactness of this range means that it is feasible to use the performance of a program on V8 to predict its performance on Nitro. Performance is much less predictable on Nitro for the other systems (Scheme2JS has slowdowns ranging from 0.8 to 22, and Spock has slowdowns ranging from 7 to 35).

For JägerMonkey the range of the slowdowns for Gambit-JS is 1.3 to 2.1 for the benchmarks in the first group, and is 5 to

8.3 for 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. 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 JägerMonkey less efficiently than by V8 and Nitro. This has been verified with a test program which creates and calls closures in a tight loop.

Because of this issue, the range of the slowdowns for Gambit-JS on JägerMonkey (1.3 to 8.3) is wide enough that it is hard to use the performance of a program on V8 to predict the performance on JägerMonkey. Nevertheless, the range of slowdowns is similar to those of Scheme2JS (1 to 5), and of Spock (2.1 to 8.6). Unfortunately, we have yet to find an implementation of closures compatible with the GVM that is as efficient on JägerMonkey as the other VMs. This would narrow the range of slowdowns for Gambit-JS and improve the quality of performance prediction.

6. Conclusion

We have proposed a VM-based approach for implementing tail calls without stack growth and first-class continuations in JavaScript. 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. We use the incremental stack/heap strategy [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 on three popular JavaScript VMs 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 our approach. When comparing Gambit-JS to Scheme2JS, Gambit-JS is 1.1 to 96 times faster on V8, 1.8 to 27 times faster on JägerMonkey, and 0.6 to 1443 times faster on Nitro. When comparing Gambit-JS to Spock, Gambit-JS is 3 to 7 times faster on V8, 1.5 to 31 times faster on JägerMonkey, and 14 to 64 times faster on Nitro.

Although this paper has focused on compiling the GVM to JavaScript, the approach has the advantage that it relies on few language constructs, so it is easy to adapt to other dynamic languages. In fact, the GVM to JavaScript translator presented in this paper is part of a *universal* back-end which also targets Python and Ruby. A large part of the back-end is common to all the target languages.

Acknowledgments

We wish to thank Florian Loitsch and Felix Winkelmann for helping us understand their systems. This work was supported by the Natural Sciences and Engineering Research Council of Canada and Mozilla Corporation.

References

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

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

[3] Chicken-Spock. 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 Notices*, 30(9):17–20, 1995.

[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, 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. ACM.

[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 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.

[9] 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.

[10] 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. ACM.

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

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

[13] F. Loitsch. *Scheme to JavaScript Compilation*. PhD thesis, Université de Nice Sophia Antipolis, 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, 1995. Springer-Verlag.

[15] D. Yoo. *Building Web Based Programming Environments for Functional Programming*. PhD thesis, Worcester Polytechnic Institute, 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 9. Source code of `fib35`

```
1.  (define (odd n)
2.    (if (= n 0) #f (even (- n 1))))
3.
4.  (define (even n)
5.    (if (= n 0) #t (odd (- n 1))))
6.
7.  (run-benchmark
8.   "oddeven"
9.   (lambda () (odd 100000000)))
```

Figure 10. Source code of `oddeven`

```
1.  (define (app lst1 lst2)
2.    (if (pair? lst1)
3.        (cons (car lst1) (app (cdr lst1) lst2))
4.        lst2))
5.
6.  (define (one-up-to n)
7.    (let loop ((i n) (lst '()))
8.      (if (= i 0)
9.          lst
10.         (loop (- i 1) (cons i lst)))))
11.
12. (define (explore x y placed)
13.   (if (pair? x)
14.       (+ (if (ok? (car x) 1 placed)
15.              (explore (app (cdr x) y)
16.                       '()
17.                       (cons (car x) placed))
18.              0)
19.          (explore (cdr x)
20.                   (cons (car x) y)
21.                   placed))
22.       (if (pair? y) 0 1)))
23.
24. (define (ok? row dist placed)
25.   (if (pair? placed)
26.       (and (not (= (car placed) (+ row dist)))
27.            (not (= (car placed) (- row dist)))
28.            (ok? row (+ dist 1) (cdr placed)))
29.       #t))
30.
31. (define (nqueens n)
32.   (explore (one-up-to n)
33.            '()
34.            '()))
35.
36. (run-benchmark
37.  "nqueens12"
38.  (lambda () (nqueens 12)))
```

Figure 11. Source code of `nqueens12`

```
1.  (define (ctak x y z)
2.    (call/cc
3.     (lambda (k) (ctak-aux k x y z))))
4.
5.  (define (ctak-aux k x y z)
6.    (if (not (< y x))
7.        (k z)
8.        (ctak-aux
9.         k
10.        (call/cc
11.         (lambda (k) (ctak-aux k (- x 1) y z)))
12.        (call/cc
13.         (lambda (k) (ctak-aux k (- y 1) z x)))
14.        (call/cc
15.         (lambda (k) (ctak-aux k (- z 1) x y))))))
16.
17. (run-benchmark
18.  "ctak"
19.  (lambda () (ctak 22 12 6)))
```

Figure 12. Source code of `ctak`

```
1.  (define (contfib n)
2.    (if (< n 2)
3.
4.        (call/cc
5.         (lambda (k)
6.           (k 1)))
7.
8.        (+ (contfib (- n 1))
9.           (contfib (- n 2)))))
10.
11. (run-benchmark
12.  "contfib30"
13.  (lambda () (contfib 30)))
```

Figure 13. Source code of `contfib30`

```
1.  (define fail (lambda () #f))
2.
3.  (define (in-range a b)
4.    (call/cc
5.     (lambda (cont)
6.       (enumerate a b cont))))
7.
8.  (define (enumerate a b cont)
9.    (if (> a b)
10.       (fail)
11.       (let ((save fail))
12.         (set! fail
13.           (lambda ()
14.             (set! fail save)
15.             (enumerate (+ a 1) b cont)))
16.         (cont a))))
17.
18. (define (btsearch n)
19.   (let* ((n*2 (* n 2))
20.          (x (in-range 0 n))
21.          (y (in-range 0 n)))
22.     (if (< (+ x y) n*2)
23.         (fail) ;; backtrack
24.         (cons x y))))
25.
26. (run-benchmark
27.  "btsearch2000"
28.  (lambda () (btsearch 2000)))
```

Figure 14. Source code of `btsearch2000`

```
1.  ;; Queues.
2.
3.  (define (next q) (vector-ref q 0))
4.  (define (prev q) (vector-ref q 1))
5.  (define (next-set! q x) (vector-set! q 0 x))
6.  (define (prev-set! q x) (vector-set! q 1 x))
7.
8.  (define (empty? q) (eq? q (next q)))
9.
10. (define (queue) (init (vector #f #f)))
11.
12. (define (init q)
13.   (next-set! q q)
14.   (prev-set! q q)
15.   q)
16.
17. (define (deq x)
18.   (let ((n (next x)) (p (prev x)))
19.     (next-set! p n)
20.     (prev-set! n p)
21.     (init x)))
22.
23. (define (enq q x)
24.   (let ((p (prev q)))
25.     (next-set! p x)
26.     (next-set! x q)
27.     (prev-set! q x)
28.     (prev-set! x p)
29.     x))
30.
31. ;; Process scheduler.
32.
33. (define (boot)
34.   ((call/cc
35.     (lambda (k)
36.       (set! graft k)
37.       (schedule)))))
38.
39. (define graft #f)
40. (define current #f)
41. (define readyq (queue))
42.
43. (define (process cont)
44.   (init (vector #f #f cont)))
44. (define (cont p) (vector-ref p 2))
45. (define (cont-set! p x) (vector-set! p 2 x))
46.
47. (define (spawn thunk)
48.   (enq readyq
49.        (process (lambda (r)
50.                   (graft (lambda ()
51.                            (end (thunk))))))))
52.
53. (define (schedule)
54.   (if (empty? readyq)
55.       (graft (lambda () #f))
56.       (let ((p (deq (next readyq))))
57.         (set! current p)
58.         ((cont p) #f))))
59.
60. (define (end result) (schedule))
61.
62. (define (yield)
63.   (call/cc
64.    (lambda (k)
65.      (cont-set! current k)
66.      (enq readyq current)
67.      (schedule))))
68.
69. (define (wait x)
70.   (if (> x 0)
71.       (begin
72.         (yield)
73.         (wait (- x 1)))))
74.
75. (define (threads n)
76.
77.   (let loop ((n n))
78.     (if (> n 0)
79.         (begin
80.           (spawn (lambda () (wait 100000)))
81.           (loop (- n 1)))))
82.
83.   (boot))
84.
85. (run-benchmark
86.  "threads10"
87.  (lambda () (threads 10)))
88.
```

Figure 15. Source code of `threads10`