Exceptional Continuations in JavaScript

Florian Loitsch

Inria Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex,
France
http://www.inria.fr/mimosa/Florian.Loitsch

ABSTRACT

JavaScript, the main language for web-site development, does not feature continuation. However, as part of client-server communication they would be useful as a means to suspend the currently running execution.
In this paper we present our adaption of exception-based continuations to JavaScript. The enhanced technique deals with closures and features improvements that reduce the cost of the work-around for the missing `goto`-instruction. We furthermore propose a practical way of dealing with exception-based continuations in the context of non-linear executions, which frequently happen due to callbacks.
Our benchmarks show that under certain conditions continuations are still expensive, but are often viable. Especially compilers translating to JavaScript could benefit from static control flow analyses to make continuations less costly.

1. Introduction

Xml-http-requests are an integral part of Ajax and the now called "Web 2.0". Basically they allow JavaScript (standardized as EcmaScript (ECMA-262 1999)) programs to interactively communicate with the server: a request is sent to a given URL and once the server returns, a callback is invoked. Both synchronous and asynchronous forms exist, but usually only the asynchronous form is usable.[1]
Due to the asynchronous nature of `xml-http-requests` programmers need to determine the work that needs to be done after the requests returns, and pack it into the callback function. Continuations could free developers from this work. Some languages (eg. Scheme (Kelsey et al. 1998)) feature first class continuations, but even simple one-shot continuations (suspend/resume) would be sufficient for our task.[2]

[1] When `xml-http-requests` are used synchronously they block the User Interface until the call is finished. A usually unacceptable behavior.

[2] An efficient `suspend/resume` implementation is also interesting for cooperative threading (see for example Fair Threads (Serrano et al. 2004)).

Proceedings of the 2007 Workshop on Scheme and Functional Programming
Université Laval Technical Report DIUL-RT-0701

JavaScript does not feature any construct similar to continuations, though. Most interpreters carry the necessary information to efficiently implement them, but as far as we know only Rhino (Mozilla Foundation) gives access to its continuations. In general continuations need to be implemented on top of JavaScript's high level constructs.
Continuation Passing Style (CPS) lends itself for this task and with sufficiently high level features (in particular closures) CPS can be implemented as a simple source code transformation (see for example (Steele 1976)). A program in CPS form, as the name suggests, passes the current continuation directly as a parameter to every function and the continuation is hence always available. This technique does indeed work in JavaScript, and some systems such as Links (Cooper et al. 2006) actually use it. In Links the continuations are not exposed to the developers either, but are used internally for threading and transparent asynchronous `xml-http-requests`.
CPS's efficiency is however largely dependent on the speed of closure creation and tail call handling. Neither are fast in current mainstream JavaScript implementations and two handwritten benchmarks (`fib` and `nested`) were 30 and 130 times slower than the native versions in our test setup. CPS transformed programs furthermore change the call convention which makes it cumbersome to interface with existing JavaScript code. We therefore looked for different techniques without these drawbacks. Our focus eventually concentrated on an exception-based technique similar to the one presented by Tao (Tao 2001) or Sekiguchi *et al.* (Sekiguchi et al. 2001). Their work does not present full fledged continuations but only a way of suspending and resuming executions. In the first part of the next section we will show how to adapt the technique to JavaScript. One of our motivations for this work was a Scheme-to-JavaScript compiler where we need full-fledged continuations to support `call/cc`. In the second part we show how we extended the given technique for this more general form of continuations. Work in this direction on a minimal language without side-effects has already been published (Pettyjohn et al. 2005).

1.1 Organization

Section 2 presents exception-based continuations and summarizes the work that has been done in this area. In Section 2.2, 2.3 and 2.4 we show how to adapt these existing techniques to JavaScript. Section 3 then shows how we can implement `call/cc` using similar techniques. In Section 4 we discuss some optimizations. Ways to handle callbacks are proposed in Section 5. Section 6 presents the result of our benchmarks. Related work is discussed in Section 7 and we finally conclude in Section 8.

2. Suspend/Resume with Exceptions

This section summarizes existing work on exception-based continuations by Tao (Tao 2001) and Sekiguchi *et al.* (Sekiguchi et al. 2001). In both cases the technique has been developed for transparent migration and checkpointing, and uses Java/JVM and/or C++ as target-language. As our work is targetted at JavaScript we will use JavaScript for all code samples, though. Also, we are going to ignore JavaScript's higher-order functions during the initial summary. A discussion of this property is delayed to Section 2.4 where we evolve the summarized technique for higher order languages.

2.1 Idea

```
1: function sleep(ms) {
2:   suspend(function(resume) {
3:             setTimeout(resume, ms);
4:           });
5: }
```

Figure 1: Sleep implemented through suspend/resume

The goal of `suspend/resume` is to save the current state of an execution and to be able to resume it later on. We propose a library function `suspend` which, similarly to Scheme's `call/cc` takes a function as parameter. `suspend` executes the given procedure in turn with a reified version of the current continuation as parameter. Independent of the outcome of this call (in particular exceptions are ignored), it then halts the execution. The program is effectively stopped, until an external event invokes the continuation to resume execution. Once resumed the continuation has served its purpose and becomes invalid, so it can not be invoked multiple times. Figure 1 shows how `suspend` would be used to create the missing `sleep` function in JavaScript. `suspend` starts by executing the anonymous function of line *2*. This procedure passes the given continuation `resume` to JavaScript's `setTimeout`, which prepares a timeout with the continuation as callback. The anonymous function then returns, and `suspend` halts the execution. After `ms` milliseconds JavaScript triggers the timeout-event and invokes the stored callback (the `resume`-continuation), which resumes the execution.

The implementation of the `suspend` function is not local and its presence requires the instrumentation of all other functions. Each function needs to be able to save its current activation frame (local variables, and current position within the code) and to restore it from this data. Contrary to CPS where the continuation is already given as parameter, we create the continuation only when the `suspend` function is called from inside the program: `suspend` raises a special exception[3] which triggers the saving in each live function. The important work during suspension is hence done by each function separately.

As expected, restoration rebuilds the call-stack by asking each function to rebuild its activation frame. Once the call-stack has been rebuilt the continuation continues normally.

Although `suspend` and `resume` are tightly coupled we present them in separate sections. This makes sense because the `suspend` code might be used independently (see Section 4.3).

2.2 Suspend

During suspension the whole call-stack needs to be saved. As JavaScript does not give access to the stack itself, all functions

[3] The exception starting the saving gives the whole technique its name, but is itself not essential. Another convention could use a special return-value to trigger the saving. Depending on the interpreter (or the JIT-compiler) this could even be faster.

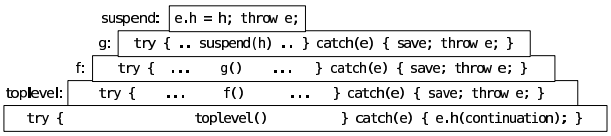


Figure 2: A global view of saving mechanism.

are instrumented so they can save their activation frame. The sum of all these activation frames represents the complete call-stack. To simplify the description (and implementation) we wrap the top-level into a function. The new top-level hence consists of a simple call to a function which contains the original top-level.

Saving is initiated by the `suspend` function when it throws a special exception. The exception triggers the saving code in each function. Figure 2 presents a high level view of the suspension mechanism. When `suspend` is called with parameter `h`, it stores the given procedure in a special exception. It subsequently throws the exception which is intercepted by the first (youngest) function on the stack. After having saved its activation frame the function then rethrows the exception. The same exception is then intercepted by the next function, until finally at the bottom a global exception handler captures the exception. The handler builds a proper continuation-object out of the saved data, and passes it to the `h`-function which had been given to `suspend`. When `h` returns the execution is halted.

```
1: function sequence(f, g) {
2:   print('1: ' + f());
3:   return g();
4: }
```

Figure 3: Running example

```
 1: function sequence(f, g) {
 2:   var tmp1;
 3:   var index = 0;
 4:   try {
 5:     index = 1; tmp1 = f();
 6:     print('1: ' + tmp1);
 7:     index = 2; return g();
 8:   } catch(e) {
 9:     if (e instanceof ContinuationException) {
10:       var frame = new Object();
11:       frame.index = index; // save position
12:       frame.f = f; frame.g = g;
13:       frame.tmp1 = tmp1;
14:       e.pushFrame(frame);
15:     }
16:     throw e;
17:   }
18: }
```

Figure 4: with suspension code

Figure 4 takes a closer look at a single instrumented function. The original non-instrumented version can be found in Figure 3. In addition to the previously mentioned `try/catch`, an `index` variable has been introduced which represents the instruction pointer. Note that only calls potentially leading to a `suspend` (dubbed "unsafe calls") need to be indexed. The call to `print` is safe, and therefore doesn't update the `index` variable. The correct identification of *safe* function calls needs be determined by static analyses (see Section 4.3). The `catch` part responsible for saving starts at line *8*. A new frame object is constructed and filled with the local variables

and the index variable. In this example the exception itself serves as container for the continuation data, and the frame object is hence stored in the exception. Finally the exception is rethrown.

2.3 Resume

At the end of suspension the global exception handler invokes the function `h` which had been given to `suspend` (see Section 2.2), but then halts the execution. The program can only resume, if an external event triggers the invocation of the continuation. Usually `h` registers the continuation (which has been given as parameter) as call-back. In the example of Figure 1 the continuation was stored as call-back for the timeout-event.

```
 1: function sequence(f, g) {
 2:   var tmp1, goto = false;
 3:   if (RESTORE.doRestore) {
 4:     var frame = RESTORE.popFrame();
 5:     index = frame.index;
 6:     f = frame.f; g = frame.g;
 7:     tmp1 = frame.tmp1;
 8:     goto = index; // emulate a jump
 9:   }
10:   ... <suspension-code omitted> ...
11:   switch (goto) {
12:    case false:
13:    case 1: goto = false;
14:            tmp1 = f();
15:            print('1: ' + tmp1);
16:            // fall-through
17:    case 2: goto = false;
18:            return g();
19:   }
20:   ... <suspension-code omitted> ...
21: }
```

Figure 5: with restoration code

When the continuation is invoked it starts by setting a global restoration flag `RESTORE.doRestore`. Subsequently it calls the last (oldest) function of the saved stack (in our case the top-level). Each function is then responsible for restoring its original activation frame, and calling the next function of the initial stack. The global flag serves as switch for restoration mode. The saved activation frame itself too is accessible through a global variable. Figure 5 shows the instrumented version of our running example (Figure 3). A test first checks if the program is in restoration or normal execution mode. In the first case it restores the values of the local variables, and jumps to the saved position. Eventually the execution arrives at the saved location and invokes the next function (still in restoration mode), which in turn restores itself and calls the next function. The restoration is finished when the `suspend` function is reached: `suspend` clears the restoration flag and returns. The execution then continues normally.
In our example a `switch`-statement was introduced to emulate the jump to the target given by the `index` variable. In general blocks are converted into `switch` statements and branching constructions are modified so they reenter the saved branch. Function calls are transformed into A-normal form (Flanagan et al. 1993), so the already calculated parameters are not executed multiple times. Figure 6 gives some examples of these transformations. Additional material can be found in (Tao 2001) and (Sekiguchi et al. 2001). The `goto` emulation makes code examples more difficult to read and we will from now use an informal "`goto index;`" form, too. Figure 7 shows the complete `sequence`-function with suspension and restoration instrumentation.

```
1: {
2:   safe1;
3:   safe2;
4:   unsafe();
5:   safe3;
6: }
```

```
1: switch (goto) {
2:   case false: // default mode.
3:                // no restoration
4:     safe1; safe2;
5:  // jump to unsafe statement
6:   case 1: goto = false;
7:     unsafe();
8:     safe3;
9: }
```

```
1: if (test) {
2:   unsafe1();
3:   unsafe2();
4: } else
5:   unsafe3();
```

```
1: if ((goto && goto <= 2) ||
2:     (!goto && test)) {
3:   switch (goto) {
4:     case 0: case 1:
5:             goto = false;
6:             unsafe1();
7:     case 2: goto = false;
8:             unsafe2();
9:   } else {
10:    goto = false;
11:    unsafe3();
12:  }
```

Figure 6: Goto examples

```
 1: function sequence(f, g) {
 2:   var tmp1;
 3:   var index = 0;
 4:   var goto = false;
 5:   if (RESTORE.doRestore) {
 6:     var frame = RESTORE.popFrame();
 7:     index = frame.index;
 8:     f = frame.f; g = frame.g;
 9:     tmp1 = frame.tmp1;
10:     goto = index;
11:   }
12:   try {
13:     switch (goto) {
14:       case false:
15:       case 1: goto = false;
16:         index = 1; tmp1 = f();
17:         print('1: ' + tmp1);
18:       case 2: goto = false;
19:         index = 2; return g();
20:     }
21:   } catch(e) {
22:     if (e instanceof ContinuationException) {
23:       var frame = new Object();
24:       frame.index = index; // save position
25:       frame.f = f; frame.g = g;
26:       frame.tmp1 = tmp1;
27:       e.pushFrame(frame);
28:     }
29:     throw e;
30:   }
31: }
```

Figure 7: with suspension and restoration code

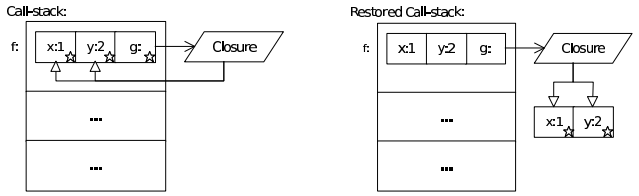
2.4 Suspend/Resume in JavaScript

The previous sections give an overview of `suspend/resume` in first-order languages like C++ or Java/JVM. JavaScript, however, is a higher order language and hence features functions as first class citizens. In this section we will discuss the implications of this property.

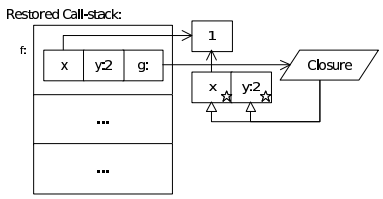
JavaScript's functions have semantics similar to Scheme procedures. That is, free variables are lexically scoped and during the creation of functions the current environment is saved in the closure. Closures contain hence references to variables of activation frames.[4] This however poses problems when the original call-stack is destroyed by a call to `suspend`. A similar call-stack is rebuilt during the `resume`, but the closure's references are still referencing variables of the old stack. The following example demonstrates such a case:

```
1: function f() {
2:     var x = 1; var y = 2;
3:     var g = function() { print(x, y); };
4:     suspendCall();
5:     x = 3;
6:     g(); // should print 3, 2
7: }
```

When the program reaches line *4* the stack structure resembles the diagram of Figure 8a. The call-stack contains a list of activation frames with `f`'s activation frame on top. The frame contains `f`'s three local variables `x`, `y` and `g`. The variable `g` points to a function which in turn captures `f`'s `x` and `y`. For explanatory purposes we have marked locations of this original call-stack with stars.
During the call to `suspendCall` the continuation mechanism throws an exception and saves the variables of all stack-frames. When the execution is resumed a similar stack is reconstructed. This new stack can be seen in Figure 8b. The restored call-stack is (as intended) similar to the original call-stack, but the closure `g` still references variables of the old call-stack. This does not pose any problem for constant variables like `y`, but is incorrect for all others. In our example the `x` of the new frame is changed, but `g` will still reference the unchanged `x` and hence incorrectly print `1`.



(a) before (b) after



(c) boxed

Figure 8: Call-stacks before and after callcc-call.

Our solution is to box all non-constant escaping variables (a conservative super set of the concerned variables). The closure will still reference an outdated variable, but the referenced box of the escaped variables will be in sync with the equivalent variables of the new stack-frame (see Figure 8c).[5]

[4] In fact, most JavaScript implementations currently just store the call-stack itself in the closure.

[5] Scheme (and other) compiler writers will not be surprised by the solution. Boxing of escaping variables is a common practice in Scheme compilers (Kranz et al. 1986), but usually for entirely different reasons.

Another subtle difficulty is introduced by pointers to functions (which could appear in C++ too). If the variable that holds the call-target is modified the call will not work as expected. Figure 9a contains an example which demonstrates how this can be a problem.

```
1: function() {
2:     var g = function() {
3:                 g = false;
4:                 suspendCall();
5:             };
6:     g();
7: }
```

(a) call-target g is modified

```
1: function() {
2:     var g = function() {
3:                 g = false;
4:                 suspendCall();
5:             };
6:     var tmp = g;
7:     tmp();
8: }
```

(b) call-target is constant

Figure 9: modified call-target

The call at line *6* depends on the local variable `g` which is modified after the invocation. During restoration `g` is correctly restored to `false` and the program then jumps to the call location. Just calling `g` again is however not possible anymore. The solution to this problem is simple. One just needs to introduce additional local variables so that calls do not depend on variables that are changed outside their scope. Figure 9b shows the corrected version.

3. Call/cc

`Suspend/Resume` is sufficient for asynchronous communication and cooperative threading. In the context of Scheme (and other languages) full fledged continuations are however needed. This section presents the changes to evolve `suspend/resume` to `call/cc`.
`Suspend/resume` basically pauses the control flow. Instead of returning, `suspend` aborts the execution until an event invokes the `resume`-continuations. With the exception of event-handling code, the program continues semantically as if no instruction had been executed between the end of the `suspend`-function and its continuation.
`Call/cc`-continuations, on the other hand, are more flexible. They can be invoked at any time and multiple times. In particular users are free to execute code between the return of `call/cc` and the invocation of the captured continuation. This raises an important question: what happens to (stack-)variables that are modified after the continuation has been captured? Semantically there are two possibilities: - either these variables are restored to the value they had when the continuation was captured; - or they should be left at their new value. Whereas the first choice could be useful for checkpointing, etc. it is the latter one which is generally adopted. Similar to Scheme we want hence modifications to variables remain when continuations are executed.
The function in Figure 10a, for instance, would yield different results depending on the chosen semantics. After the first invocation of the continuation the `print` in line *4* should obviously print `1`, but more importantly (due to the assignment in the following line) other invocations could then either continue printing `1` (value at time of suspension) or could then print `2`, `3`, etc. We would like our technique to print the incrementing sequence, but our previous suspension technique ignores modifications that happened to

```
1: function () {
2:   var x = 1;
3:   callccCall();
4:   print(x);
5:   x = x + 1;
6: }
```

(a) call/cc example

```
1: function () {
2:   // restoration code
3:   // producing 'frame'
4:   ...
5:   callccCall();
6:   print(x);
7:   x = x + 1;
8:   if (frame)
9:     frame.x = x;
10: }
```

(b) update at the end

```
1: function () {
2:               var x;
3:               var frame = false;
4:               if (RESTORE.doRestore) {
5:                 frame = RESTORE.popFrame();
6:                 index = frame.index;
7:                 x = frame.x;
8:                 goto index;
9:               }
10:              try {
11:                x = 1;
12: gotoTarget1:   callccCall();
13:                print(x);
14:                x = x + 1;
15:              } catch (e) {
16:                /* save-code */
17:              } finally {
18:                if (frame) { frame.x = x; }
19:              }
20: }
```

(c) complete version with update in finally

Figure 10: Call/cc with side-effects

stack-variables after the continuation has been saved. It would have printed 1 all the time: as `x` does not escape it is not boxed (see Section 2.4) and during the construction of the continuation `x` is saved with value 1. Every restoration of the original call-stack would subsequently restore this value. There are (at least) two ways to obtain the chosen behavior. One can either box all muted variables or track the changes and update the continuation. Boxing is easier to implement but for efficiency reasons we use the second technique. Suppose `frame` is the name of the continuation structure that holds all local variables. We could then just add a new line updating the continuation at the end of the function as in Figure 10b.

If the function has already been suspended once, then the `frame` variable is not `false` and the value for `x` is updated in line *9*. The program now correctly outputs 1, 2, etc. In general just updating at the end of the function is however incorrect. There are many means to exit a function, and only few go through the last line of a function. We therefore use a `finally` clause of a `try/catch` (which incidentally has to be used for continuation support anyway). This way variables are always updated before leaving the function. The complete version (still without suspension code) can be found in Figure 10c.

4. Suspend/Resume and Call/cc Optimizations

This section presents some optimizations to the previously presented implementation. All important technical aspects have already been discussed in previous sections, and we will hence only focus on implementation and efficiency issues. The techniques shown in this section do not add any functionality but succeed in reducing the modifications to the original source code thus making the code lighter and faster. In the remainder of the chapter we will use `call/cc` and `suspend/resume` indifferently as all optimizations apply for both scenarios.

We will first present our hoisting- and tail-call optimizations in separate sections and then discuss miscellaneous optimizations in the following section.

4.1 Hoisting Instructions

During the restoration of the call-stack the program needs to execute a jump to the saved target which is an expensive operation in JavaScript. The following optimization moves the targets to later locations (thereby skipping instructions) which generally reduces the cost of the jumps. The skipped instructions are duplicated at the jump-origin, and are executed before the jump.

As discussed in Section 2.3 JavaScript does not feature any `goto`-instruction, and the body of functions has to be transformed to emulate jumps. Most constructs surrounding a jump-target need to test if they are executing an emulated `goto` or if they are executed normally. We therefore define the cost of a jump-target to be its nesting-level. The more nested a target is, the more it costs (even in normal operation, as the tests have to be done all the time).

The goal of this optimization is to reduce the cost of jump-targets. It basically copies code from the jump-target to the jump-origin. The copied instructions are then already executed before the jump, and the jump-target can be advanced so it skips the copied code. The moved jump-target might leave constructs, thereby reducing the nesting-level, and as a result one could avoid its instrumentation.

In Figure 11 we demonstrate on an example the impact this optimization can have. In the first code-snippet we informally state the need for a jump to the unsafe call (labeled with “target:”). The second code-sample shows the expensive transformations needed to emulate this `goto`-instruction. As the unsafe call is embedded in a `while`-loop and an `if`, both constructs need to be transformed for the emulated jump. In Figure 11c we copied the targets statement to the jump-origin in line *4*. The call of line *15* is therefore already executed before the jump, and the jump-target has been advanced to the instruction following the call. The `if`-statement is finished, and the next instruction would thus be the test of the `while`-loop (line *9*) which is equivalent to the `while`-construct itself. The new jump-target is hence just before the `while`-statement. Both the loop and the `if`-statement do not contain any jump-targets anymore and can hence be left untransformed (Figure 11d).

The suspension code is left nearly untouched. The sole change rectifies the scope of the suspension `try/catch`. As the restoration code now contains calls to unsafe locations too, it is necessary to enlarge the exception-handler so that the `try`-keyword is before the `if`-statement in the beginning of the function.

Due to implementation-specific reasons we currently restrict the copied code to be at most the targeted call and a potential assignment of its return-value. In the future we would like to remove this limit and experiment with bigger copies.

Concluding this section we would like to point out that this optimization is not always beneficial. Blocks containing jump-targets are generally transformed to `switch`-statements, with one exception: when there is only one jump-target and the target is (part of) the first statement. In this case the block can be left untouched. The presented optimization however advances jump-targets, and could move the target from the first statement to the second statement. In this case the previously untouched block would then be transformed into a `switch`-statement. Our implementation does not yet take into account this special case.

```
if (RESTORE.doRestore) {
  ...
  goto target;
}
print('before');
while (test1) {
  print('loop');
  if (test2) {
    doSomething;
  } else {
    print('if');
target: unsafeCall();
  }
}
print('after');
```

(a) goto-example

```
if (RESTORE.doRestore) {
  ...
  goto = 1;
}
switch (goto) {
case false:
  print('before');
case 1:
  while (goto == 1 || test1) {
    switch (goto) {
    case false: print('loop');
                // fall through
    case 1:
      if (goto == 0 && test2)
        doSomething;
      else
        switch (goto) {
        case false:
                print('if');
              // fall through
        case 1: goto = false;
                unsafeCall();
        }
    }
  }
  print('after');
}
```

(b) jump to the call itself

```
 1: if (RESTORE.doRestore)
 2: {
 3:   ...
 4:   unsafeCall();
 5:   goto target;
 6: }
 7: print('before');
 8: target:
 9: while (test1) {
10:   print('loop');
11:   if (test2) {
12:     doSomething;
13:   } else {
14:     print('if');
15:     unsafeCall();
16:   }
17: }
18: print('after');
```

(c) unsafe call copied to jump-origin

```
if (RESTORE.doRestore) {
  ...
  goto = 1;
  switch (goto) {
    1: unsafeCall(); break;
  }
}
switch (goto) {
case false:
  print('before');
case 1: goto = false;
  while (test1) {
    print('loop');
    if (test2) {
      doSomething;
    } else {
      print('if');
      unsafeCall();
    }
  }
  print('after');
}
```

(d) with goto-emulation

Figure 11: jump to before and after the call.

4.2 Tail-call Optimization

The continuation-technique which has been presented until now ensures that a restored call-stack is similar to the original one. Like the original stack the restored stack would have the same number of activation frames and each activation frame would have the same values as the original stack. Often not all of these frames are still needed, though. Suppose `f` calls `g` which in turn tail-calls `h`. When the continuation has been captured during `g`'s tail-call, then the restoration could skip `g` if `f` called `h` directly. In practice changing the `f`'s call target is however not that easy. Each function restores itself and is then responsible to reexecute the *same* call as it had done when it was suspended. Function `f` would hence restore itself and then reexecute the call to `g`. In the optimized version `f` should call `h` (a function which might not even be visible to `f`) directly to skip `g`.

In this section we continue evolving our continuation technique so that a tail-call optimization becomes possible. In the new version functions save a pointer to themselves in addition to their activation-frame data. This pointer is then looked up during restoration to retrieve the next function. Tail-calling functions are simply skipped during saving and are hence removed in the restored stack. In our running example, the tail-calling `g` would not save its frame and would hence not appear in the saved continuation data. During restoration functions must not just call the same next function as before, but have to retrieve the function pointer in the next frame. The function `f` would retrieve `h`'s frame (as `g` did not register its frame), and therefore call `h` as next function.

The benefits of this optimization are twofold: the number of activation frames is reduced, and the instrumentation for tail-call locations is simplified. Indeed, tail-calling functions will not be restored, and it is hence unnecessary to add jump-emulations to their tail-call locations.

```
 1: function sequence(f, g) {
 2:   var tmp1, index = 0, isTail = false;
 3:   if (RESTORE.doRestore) {
 4:     var frame = RESTORE.popFrame();
 5:     index = frame.index;
 6:     f = frame.f; g = frame.g;
 7:     tmp1 = frame.tmp1;
 8:     // restore remaining stack:
 9:     var callCcTmp = RESTORE.callNext();
10:     switch (index) {
11:       case 1: tmp1 = callCcTmp; break;
12:     }
13:     goto index;
14:   }
15:   try {
16: index = 1;     tmp1 = f();
17: gotoTarget1:   print('1: ' + tmp1);
18: isTail = true; return g();
19:   } catch(e) {
20:     if (e instanceof ContinuationException &&
21:         !isTail) {
22:       var frame = new Object();
23:       frame.index = index; // save position
24:       frame.f = f; frame.g = g;
25:       frame.tmp1 = tmp1;
26:       e.pushFrame(frame, this, arguments.callee);
27:     }
28:     throw e;
29:   }
```

Figure 12: complete version with assignment in restoration code.

Figure 12 shows the new (suspension and restoration) code of the `sequence`-example (Figure 3). We will first focus on the suspension code, and hence skip the restoration-`if` for now. As already mentioned in the summary a pointer to the currently running function is saved during suspension. In JavaScript this pointer is readily available as a combination of the `this`-keyword and the `arguments.callee` (line *26*). The tail-call optimization itself can be seen in line *18* and line *21*. Tail-calls are now specially marked (the `isTail`-variable is set to `true`), and if a function is tail-calling then it skips itself during saving (due to the test in line *21*). Tail-calling functions are hence ignored during saving, and it is the restoration part's responsibility to determine and execute the next non-skipped function. The necessary information is inside the

saved continuation through the means of the function-pointers. Instead of calling the same previous call as before one just has to retrieve the next function of the continuation-state and invoke it instead. In order to clarify the code we hide this operation and use a method-call (`RESTORE.callNext`) in line *9* instead.
The result is then assigned to their respective variables (if any). Thanks to the hoisting-optimization this task is simplified. The original call in the body is left untouched, and the copied call is simply replaced by the temporary variable `callCcTmp` which holds the result of the `callNext`-call: whereas we previously would have had `tmp1=f();` in line *11*, we now have `tmp1=callCcTmp;`.
We want to point out that this optimization is not a substitute for (expensive) proper tail-recursion handling (as presented in (Loitsch and Serrano 2007)). Frames are only discarded when continuations are taken or invoked, which is clearly not sufficient for proper tail-recursion (as required for Scheme). The main-benefit is hence not the removal of the frames but the removal of instrumentation for tail-calls. In some cases the optimization can avoid the complete instrumentation for functions: if a function's unsafe calls are only at tail-call-locations, then it does not need any instrumentation at all.

4.3 Miscellaneous Optimizations

This section groups several optimizations that are either too small to merit a separate section, or are not yet sufficiently explored (and hence subject for future work).
As `call/cc` is usually only present in few locations, most calls do not (and often even can not) reach any `call/cc`. An optimizing compiler should hence use standard compilation techniques (Muchnick 1997) to reduce the number of unsafe call-locations. We dub "unsafe" call-location calls that might eventually reach a `call/cc`. In JavaScript one can modify global variables through several ways, most of which are difficult to detect (amongst others the `eval`-function, and the global `this`-object). JavaScript itself is hence difficult to optimize in this area. Even though it is generally possible to mark most local functions as safe, calls to global functions need to be considered unsafe. However, when JavaScript is used as compilation target for a different language, then such an analysis can be often much more effective. We have implemented an ad-hoc analysis in our Scheme-to-JavaScript compiler (SCM2JS). Even though Scheme is highly dynamic this analysis was able to detect the absence of continuations in 10 out of 11 benchmarks. The remaining benchmark (a meta circular interpreter making heavy use of higher-order functions) had about 75% of its functions instrumented.
One should also consider handling functions by hand. Especially libraries are possible candidates for this special treatment. If the library itself does not use continuations then only exported higher-order functions need to be instrumented. To keep libraries generic we usually export both versions of these functions (one uninstremented and one instrumented). As a bonus even continuation-heavy programs might prefer calling the uninstremented function when they can prove that the sent parameter does not invoke `call/cc`. In SCM2JS important functions like `for-each`, `map` and others have been implemented this way. The closures sent to these functions are often small anonymous lambdas that can be easily analyzed.
In a similar vein it can be beneficial to create versions without restoration-code (and hence without `goto`-emulation). This version is sufficient in all but one context: during stack restoration the full version is needed. The switch to full version can happen at several occasions: during saving a function might save the full version instead of itself; or the `callNext` method of the `RESTORE`-object could translate the original version to the full version and call the latter. Even without this optimization the code growth is already extensive and we therefore have not yet implemented this technique. Initial tests on `tak` (one of our benchmarks) showed potential, though. The new version was about twice as fast as the old one.

5. Callbacks

Using our `call/cc`-framework the top-level is responsible for catching the suspension-exception. In web-browsers, callbacks occur however outside the dynamic extent of the original top-level. A `call/cc` inside a callback would hence fail. In this section we review the importance of callbacks, and discuss our solution to this issue.
JavaScript is usually used in web-browsers where it is responsible for the user-interface. Web-browsers provide a (mostly) standardized way, the Document Object Model (DOM) (Hors et al. 2000), for accessing visual elements through JavaScript. A recurring pattern involves the use of callbacks to react to events. Callbacks are functions stored in the DOM which are then invoked when an event occurs. Another form, not involving the DOM, can be found in Figure 1 where the timeout-callback had been used to resume the execution of the suspended program. As already mentioned in Section 2.2 suspended programs can only be awaken through external callbacks. Due to the ubiquity of callbacks it is hence important to allow `call/cc` to work inside callbacks with minimal effort for the programmer.
The solution we adopted is completely transparent. The `call/cc`-exception handler signals through a global flag `CALLCC.handler` its presence (or absence). Every function starts by testing this flag. If the handler is present the execution continues normally. If the flag is not set, though, then the function is not inside the dynamic extent of a `call/cc` exception handler. In the latter case the function creates the exception handler itself before continuing.

```
 1: function callCcHandler(f, f_this, args) {
 2:   try {
 3:     CALLCC.handler = 'present';
 4:     f.apply(f_this, args);
 5:   } catch (e) {
 6:     ...
 7:   } finally {
 8:     CALLCC.handler = 'absent';
 9:   }
10: }
```

Figure 13: `callCcHandler` creates a `call/cc`-exception handler.

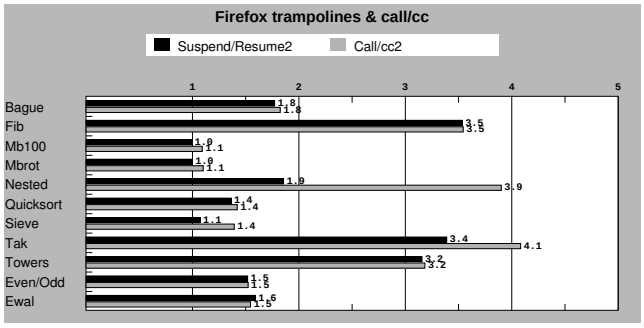
To avoid code duplication we have implemented the function `callCcHandler` (Figure 13) which contains the `try/catch` originally found in the top-level. It takes a function as parameter, and invokes it inside the `try/catch`. Functions that are invoked through the handler, are hence inside a dynamic extent of a `call/cc`-exception handler.
Initially `CALLCC.handler` is set to `'absent'`. The first function that is executed will hence encounter the `absent`-state and therefore execute the `callCcHandler` function. `callCcHandler` creates a `try/catch` and sets `CALLCC.handler` to `'present'`. The following functions are then protected and do not need to call the handler again. When the top-level is left, `callCcHandler` sets `CALLCC.handler` to `absent` again. Callbacks that occur afterward encounter the `'absent'`-state again and will hence reinvoke the `callCcHandler`.
Figure 14 shows the few lines that are added to each function. If the function has been invoked outside the dynamic extent of a continuation-try/catch (as it happens for the top-level or in the case

```
1: function f(...) {
2:   if (CALLCC.handler === 'absent') {
3:     return callCcHandler(arguments.callee,
4:                          this,
5:                          arguments);
6:   }
7:   ...
```

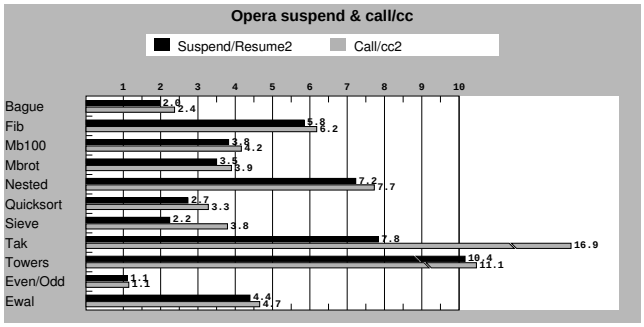
Figure 14: `Try/catch` is triggered on demand.

of callbacks) then the function invokes `callCcHandler` which creates an exception-handler. The function `callCcHandler` would set `CALLCC.handler` to `'present'` and invoke the given function. The variables `arguments.callee` (a pointer to the running function), `this` and `arguments` contain enough information to restart the function.

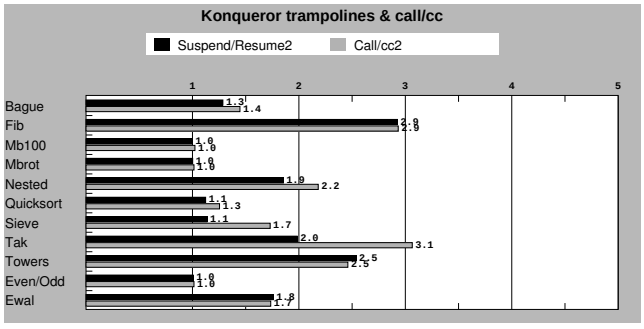
6. Benchmarks



(a) Firefox



(b) Opera



(c) Konqueror

Figure 15: Impact of suspend/resume and call/cc instrumentation. Raw code is the 1.0 mark. Lower is better.

Exception-based continuations instrument the original code and thus slow down the program even when continuations are never used. The impact however is largely dependent on the given program. A sequential program without function calls is nearly unaffected, whereas small functions with many function calls are significantly slowed down. A static analysis (like that of Shivers (1991)) is usually able to reduce the number of instrumented functions, but if continuations are heavily used then such an analysis does not help either. Links (Cooper et al. 2006), for instance, uses continuations to implement threading. Due to the huge number of possible suspension points, nearly all functions must be instrumented. When, on the other hand, continuations are used to simulate synchronous client-server communication on top of asynchronous `xml-http-requests` then only functions reaching these requests need to be modified. In this case the penalties due to continuations are furthermore usually insignificant compared to the time spent on the communication itself.

Our benchmarks are intended to measure realistic worst-case scenarios for the latter use-case. In particular we are not interested by the cost of the actual continuation-construction and -invocation but we want to determine the slow-down due to the instrumentation (even when not reaching any `call/cc`).

We have added continuation support to SCM2JS, our Scheme-to-JavaScript compiler (Loitsch and Serrano 2007). As a typing pass eliminated all instrumentation for all but one (`ewal`) benchmark we modified the original benchmarks to disturb the typing algorithm. The benchmarks still make no use of the continuation support, but the typing pass is not able to prove this anymore. As a result most (but not all) functions are now instrumented. We left our inlining pass activated too, which reduces the stress on very small functions.

To evaluate the impact of continuation instrumentation we ran our benchmarks under three Internet browsers:

- Firefox 2.0.0.3,
- Opera 9.20 build 638, and
- Konqueror 3.5.7

All benchmarks were run on an Intel Pentium 4 3.40GHz, 1GB, running Linux 2.6.21. Each program was run 5 times, and the minimum time was collected. The time measurement was done by a small JavaScript program itself. Any time spent on the preparation (parsing, precompiling, etc.) was hence not measured. The results are shown in Figure 15.

We have noticed tremendous differences between the three browsers. Konqueror seems to be the least affected, but as it was not very fast in the beginning, the time penalties are important. Opera's behavior largely depends on the benchmarks, but one can see that continuation support can be expensive. Even though Firefox has worse values than Konqueror one should note that Firefox was up to ten times faster than Konqueror. Compared to the uninstrumented version continuation-enabled code was however up to 4.1 times slower.

Despite these apparently bad results we think that continuations are viable, as most benchmarks have been modified to exhibit worst case scenarios. Even the most realistic benchmark (`ewal`) represents a non-optimal example for exception-based continuations. Its high number of anonymous functions and closures makes it difficult to analyze.

7. Related Work

Our work is an adaption and evolution of the suspension and migration techniques presented in Tao's thesis (Tao 2001) and Sekiguchi *et al.*'s paper (Sekiguchi et al. 2001). Pettyjohn *et al.* later extended this technique for `call/cc` (Pettyjohn et al. 2005) and formally

showed the correctness of their approach on a minimal language without side-effects.
Several other projects implemented continuations in JavaScript using different techniques: Narrative JavaScript (Mix) and djax (Friedlander) both unnest all constructs and explicitly handle the control-flow. Code is within a `while(true)`-loop and a `switch`-statement. Narrative JavaScript stores local variables in an object, whereas djax creates a closure at each invocation. In the latter case all local variables are declared outside the scope of the invoked function, and are thereby captured.
jwacs (Wright) and Links (Cooper et al. 2006) both use CPS to implement continuations.

8. Conclusion

We have presented exception-based continuations for JavaScript. Starting with an implementation of `suspend/resume` for C++, Java or JVM we have adapted the technique to JavaScript. We have then extended the technique to `call/cc`. We have presented several optimizations that, most of which reduce the cost of the `goto`-emulation. Finally we have discussed our implementation to deal with non-linear execution as happens with callbacks.
Our benchmarks show that full fledged continuations can still be expensive, but are now usable in many scenarios. Especially when using JavaScript as target-language, static analyses can help improving the speed of exception-based continuations.

References

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. submitted to ICFP 2006, URL `http://groups.inf.ed.ac.uk/links/papers/-links-icfp06/links-icfp06.pdf`, 2006.

ECMA. *ECMA-262: ECMAScript Language Specification*. Third edition, 1999.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

Hamish Friedlander. djax. URL `http://djax.mindcontrol-dogs.com/`.

A. Le Hors, P. Le Hegaret, G. Nicol, J. Robie, M. Champion, and S. Byrne (Eds). "Document Object Model (DOM) Level 2 Core Specification Version 1.0". W3C Recommendation, 2000.

R. Kelsey, W. Klinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: an optimizing compiler for scheme. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 219–233, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-197-0.

Florian Loitsch and Manuel Serrano. Hop client-side compilation. In *TFP 2007: Draft Proceedings of the 8th Symposium on Trends in Functional Programming*, April 2007.

Neil Mix. Narrative javascript. URL `http://neilmix.com/-narrativejs/`.

Mozilla Foundation. Rhino. URL `http://www.mozilla.org/-rhino/`.

S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming, ICFP 2005*, September 2005.

Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling. *Lecture Notes in Computer Science*, 2022:217+, 2001.

Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-819-9.

Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, May 1991.

Guy L Steele. Lambda: The ultimate declarative. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

Wei Tao. *A portable mechanism for thread persistence and migration*. PhD thesis, University of Utah, 2001. Adviser-Gary Lindstrom.

James Wright. jwacs. URL `http://chumsley.org/jwacs/-index.html`.