Hop Client-Side Compilation

Florian Loitsch and Manuel Serrano

INRIA, Sophia Antipolis, France
{florian.loitsch,manuel.serrano}@sophia.inria.fr

Abstract

Hop is a new language for programming interactive Web applications. It aims to replace HTML, JavaScript, and server-side scripting languages (such as PHP, JSP) with a unique language that is used for client-side interactions and server-side computations. A Hop execution platform is made of two compilers: one that compiles the code executed by the server, and one that compiles the code executed by the client. This paper presents the latter.

In order to ensure compatibility of Hop graphical user interfaces with popular plain Web browsers, the client-side Hop compiler has to generate regular HTML and JavaScript code. The generated code runs roughly at the same speed as hand-written code. Since the Hop language is built on top of the Scheme programming language, compiling Hop to JavaScript is nearly equivalent to compiling Scheme to JavaScript. The compiler we have designed supports the whole Scheme core language. In particular, it features proper tail recursion. However complete tail recursion optimization may slow down the generated code and is hence disabled in Hop. Most of our benchmarks were unaffected by the transformation but in the worst case programs were more than two times slower with it enabled.

The techniques presented in this paper can be applied to most strict functional languages such as ML and Lisp.

1 INTRODUCTION

Hop [11] is a new functional language designed for programming Web 2.0 applications. It is tuned for programming interactive graphical user interfaces for the Web. A Hop application executes simultaneously on two computers: one for computing the logic of the application, which we refer to as the *server* or *broker* (conforming to existing practice [16]) and one for running the graphical user interface, which is henceforth denoted as the *client*. The Hop execution model is distributed but a Hop program is made of one unique source code. Inside that code, a syntactic construction introduces broker code, another one specifies client code. Compiling a Hop program involves two different compilation processes. The broker code is compiled to native code by a compiler that has already been described in various papers [12,10]. The client code is compiled to JavaScript which is the natural language for programming graphical user interfaces on the Web. This paper describes that compilation. In Section 1.1, we shortly present the Hop programming language by an example. Then, in Section 1.2, we present the main characteristics of the client code compilation.

1.1 Hop at a glance

The following code snippet shows a small Hop program that mimics the famous *Google suggest* application: given the first characters of the entered search term popular completions are proposed.

```
1: (let ((def (<DIV> ""))
2:       (svc (service (w)
3:              (<P> (sql-exec db
4:                           "SELECT * FROM dict WHERE (prefix=~a)"
5:                           w)))))
6:    (<HTML>
7:      (<INPUT> :onkeyup
8:               ~(set! $def.innerHTML ($svc this.value)))
9:      "The definitions are:"
10:     def))
```

Basically a page (starting at line *6*) is constructed and sent to the client. It contains a `<DIV>` area (named `def`) and a text field, which reacts on `onkeyup`-events. At each event it calls the service[1] `svc` on the broker, and updates `def`. The function `svc` (declared at line *2*) executes a database query and returns the result.
Note that (except for the database query) both server and client are written in Scheme, and that switching from one to the other can be done using only one character. Client code is introduced by a `~` (tilde) and one can escape back to server-code using `$`. This construct strongly resembles Scheme's *quasiquotes* in that `$` escaped expressions are already evaluated during elaboration before sending the page to the client. During that elaboration stage the reference to `def` is then transformed to JavaScript code retrieving the `div`, and the service is transformed into a server call. The following example further demonstrates this property:

```
1: (let* ((x 0)
2:        (svc (service ()
3:               (set! x 1))))
4:   (<HTML>
5:     (<BUTTON> :onclick ~(begin ($svc) (alert $x)))))
```

Since the elaboration of this site has replaced `x` with its actual value `0`, the modification in the service has no effect on the client side and the alert shows `0`. Even though the service-call in line *5* modifies the variable `x` the program will alert `0`. During elaboration of the site, `x` had already been replaced with `0` and the modification in the service is not transmitted to the client anymore.
The Hop server associates URLs to programs. Hence, in order to start a Hop program one has to direct his Web browser to one of these URLs. This starts the execution of the program on the server. In general, web based programs are event-based, and implement the following pattern: the program is started and the server elaborates a response which is sent to the client. That response is usually made of

[1] The `service` form creates a function that can be invoked by both client and server code.

a data structure implementing an HTML element representing the graphical user interface. Once the client has received its graphical user interface it interacts with the user and, when necessary, invokes other services on the server.
One should note that while server code and client code are expressed in the same language they are intended for different purposes. The server code can access all resources of the server computer. In particular, it can access the file system, the network interfaces, or it can execute long lasting CPU intensive computations. However, it is not knowledgeable of any characteristics of the graphical user interface that are only known to the client code. The client code on the other hand knows everything about the graphical user interface but, for security reasons, has no access to other resources. This dichotomy between server code and client code is reflected by two different APIs that are available to the server and to the client.
We finish this Section by summarizing the main characteristics of Hop. *(i)* Hop is a functional language built on top of the Scheme programming language [8] with which it shares most of its syntax. *(ii)* Server code and client code are expressed in the same language. *(iii)* The tilde sign `~` introduces client code and that the dollar sign `$` inside client code escapes back to server code. *(iv)* A *service* is a function defined on the server (line *2*) that can be invoked from the client (line *8*). *(v)* Finally service invocations involve transmitting and receiving complex values that can be any compound data structure.
This section has presented a short overview of the Hop programming language. A more complete presentation can be found in [11] or at the URL of the project[2].

1.2 Compiling Hop client code: the Scm2Js compiler

```
1: (define (server-info) (string-append (host-name) " " (date)))
2: (<HTML>
3:   (<BUTTON> :onclick
4:             ~(f $(server-info)))
5:   (<SCRIPT>
6:     ~(define (f val) (alert val))))
```

FIGURE 1: Hop program example.

We have developed a compiler, named SCM2JS, to compile Hop client code to JavaScript. Hop server code is compiled by another compiler and in Figure 1 only line *4* and line *6* are hence of interest. Hop extracts these lines and sends the list of expression to SCM2JS. As can be seen, Hop client side code resembles Scheme. In fact Hop client code is a superset of IEEE Scheme [8] with one exception: it does not support exact arithmetic. Most Hop extensions consist of additional library functions or new syntactic forms that are macro-expanded be-

[2] http://hop.inria.fr.

fore the compilation takes place. The example however demonstrates some additional difficulties: SCM2JS has to deal with opaque objects (the call to the server, `$(server-info)`, is server-code and has to be treated as a black box), out-of-order compilation (the function `f` is defined in a line following the first use of `f`), and the use of unbound variables (like `alert`).
When compiling Hop client code SCM2JS allows unbound variables, and both symbol-related difficulties are hence avoided. Opaque objects are straight-forward to implement and from this point on Hop client-side compilation is mostly equivalent to a Scheme-to-JavaScript compilation. In consequence, all the techniques presented in this paper would equally apply to a pure Scheme-to-JavaScript compiler. By extension, most of the material presented here could also be useful for compiling other strict functional languages (e.g., ML) to JavaScript. In the rest of this paper we will indiscriminately use the terms "Hop client code" or "Scheme" for denoting the input language of SCM2JS.
Hop client code compilation has to fulfill two requirements:

- CPU intensive parts of Hop programs are executed on servers. However, in order to let GUIs be as reactive as possible it is important to make the Hop client code as efficient as possible. We consider of prime importance to guarantee that Hop imposes no performance penalty in comparison with traditional Web development kits whose client code is implemented in JavaScript. That is the performance of compiled Hop client code must be on par with equivalent handwritten JavaScript code. We consider performance as a potential issue even though we have noticed tremendous differences of performance depending on the hardware architecture and the JavaScript interpreter used for testing. For instance, we have found that running JavaScript programs within Firefox is nearly ten times faster than running the same programs within Konqueror. This tends to demonstrate that most users are not paying much attention to performance. Developers, on the other hand, are more concerned with performance, and a noticeable slower client side code is not acceptable.

- Scheme and JavaScript must be tightly integrated. That is all global bindings should be easily accessible from both languages, and data structures must be usable indifferently in both language. Function calls should have always the same syntax, independently where the call target has been created.

1.3 Main Contributions

From a practical point of view the main contribution of this work is the creation of a fully functional efficient Scheme-to-JavaScript compiler. Without complete tail-recursion the compiled code is on par with hand-written code, and is hence suitable for daily work.
From a technical point of view we suggest improvements to existing tail call techniques. Proper tail recursion does not exist in JavaScript and must hence be coded

by hand. We advertise the use of JavaScript's `this`-keyword to adapt existing trampoline techniques so they become compatible with existing JavaScript code (Section 4.1). We also propose an optimization to the tail recursion mechanism that allowed us to remove 40% of the tail call instrumentation in our benchmarks (Section 4.2).

1.4 Organization of the paper

Section 2 shows how SCM2JS compiles Scheme's core language to JavaScript. In Section 3 we discuss the function compilation. This specifically includes our `while` transformation for recursive loop functions. This transformation always improves the performance. The compilation of the remaining tail calls is then presented in Section 4. This transformation has no impact on most benchmarks but, in the worst case, can slow down the execution by more than a factor of 2. Section 5 shows the results of our benchmarks. Related work is discussed in Section 6. Section 7 provides the download locations of this project, and we finally conclude this paper in Section 8.
Our compiler supports fullflegded continuation, but their compilation is too complex and extensive to fit into this paper and will be the subject of another publication.

2 CORE COMPILATION

This section introduces the compilation of the Scheme core language. Function compilation and proper tail call handling are discussed in Sections 3 and 4.
JavaScript has been inspired by Scheme, and both languages are hence similar in many respects. Like Scheme, JavaScript treats functions as first class citizens and uses automatic memory management. SCM2JS is hence freed from the burden of implementing closures or a garbage collector. Moreover, many Scheme constructs can be naturally mapped to semantically equivalent JavaScript counterparts. Most transformations are as simple as transforming an array to a list. Variable argument functions, for instance, use arrays to pass the variables in JavaScript, but expect lists in Scheme. A compiled variable argument function simply copies the members of the given array into a list.
Despite the similarities compiling Scheme to JavaScript can not be accomplished by a mere source-to-source transformation. Peculiar JavaScript scoping rules and the demand for optimizations require the construction of a true abstract syntax true. JavaScript and Scheme do not share the same data types either. JavaScript, for instance, does not have any list data type and SCM2JS therefore compiles Scheme lists to instances of a new class `sc_Pair` which is part of the SCM2JS runtime system. In fact only Scheme's booleans, procedures and numbers (to a certain extent)[3] are semantically compatible with their respective counterparts in JavaScript.

[3] JavaScript numbers are floating point only. Scheme usually offers exact numbers (integers) too.

The remaining types either behave differently or do not have any corresponding JavaScript type:

- JavaScript strings are, contrary to Scheme strings, immutable. This restriction is not very limiting and users often prefer the ease of interfacing with JavaScript over a correct string representation. Depending on a compiler flag SCM2JS can either directly compile Scheme strings to JavaScript strings (thereby simplifying the interface between JavaScript and Scheme code), or translate Scheme strings to JavaScript objects of class `sc_String`. Instances of this class represent mutable strings by holding one of JavaScript's immutable strings and transparently replacing it when necessary.

- Symbols are mapped to JavaScript strings. If SCM2JS is configured for mutable strings, then JavaScript strings are unused and hence free to use as symbols (which are also immutable). Otherwise Scheme strings and symbols are both compiled to JavaScript strings, and symbols are prefixed by a special unused Unicode character in order to distinguish them from strings.

- Pairs and characters are both compiled to JavaScript objects (respectively of class `sc_Pair` and `sc_Char`). The empty list is represented by `null`.

- Vectors are mapped to JavaScript `Array`s.[4]

Due to the high level of JavaScript many standard optimizations are difficult to implement within SCM2JS. It is for instance not easy to take advantage of a typing pass. JavaScript itself is dynamically typed and does not offer any means to annotate variables with typing information. The lack of a `goto` statement too, rules out other common optimizations [13]. On the other hand the optimizations that are still applicable can have a big impact on performance. For instance, our inlining pass (modeled after [10]) was able to cut the execution time of some benchmarks in half. Inlining library functions (like `+`, `-`, etc.) proved to be even more important. Our benchmarks were up to 25 times faster with this optimization enabled. Other optimizations include hoisting of constant assignments (especially function creations) or constant propagation.

3 FUNCTION COMPILATION

Scheme procedures and JavaScript functions are very similar and a naive compilation would be straightforward. Scheme, however, makes more extensive use of procedures than JavaScript. In particular, it promotes the use of tail-recursive functions as loops. Using recursive tail calls as loops is only possible if they do not consume any stack (called "proper tail recursion"). Currently all important JavaScript interpreters are known not to perform tail call optimization and SCM2JS

[4] Despite being called "Array", this data-type is an object and consists, like all JavaScript objects, of a hashtable.

needs to handle tail calls by itself. A loop optimization pass transforms most recursive tail calls into loops. It is presented in the remainder of this Section. An optional transformation (Section 4) limits the call stack size for the remaining tail calls.
In Scheme nearly all loops are implemented as recursive tail calls. The following example demonstrates a typical loop pattern:

```
1: (let loop ((x 0)
2:            (y 0))
3:   (if <test>
4:       <body1>
5:       (begin
6:          <body2>
7:          (loop (+ y 1) x))))
```

Whenever SCM2JS encounters a tail call to the surrounding function it compiles this pattern into a `while` loop as in figure 2.

```
 1: var x = 0, y = 0;
 2: while (true) {
 3:   if (<test>) {
 4:     <body1>;
 5:   } else {
 6:     <body2>;
 7:     var tmp = y + 1;
 8:     y = x;
 9:     x = tmp;
10:     continue;
11:   }
12:   break;
13: }
```

(a) unoptimized

```
1: var x = 0, y = 0;
2: while (!<test>) {
3:   <body2>;
4:   var tmp = y + 1;
5:   y = x;
6:   x = tmp;
7: }
8: <body1>;
```

(b) optimized

FIGURE 2: Unoptimized and optimized `while` compilation of recursive loops.

Such naive source-to-source translations are only sufficient as long as loop variables are not captured. As the transformation reuses loop variables during each iteration explicit closure handling becomes necessary. The following example demonstrates this issue:

```
1: (let loop ((x 1))
2:   (store! (lambda () x))
3:   (loop (+ x 1)))
```

In this code snippet the loop variable `x` is captured by anonymous functions in line *2*. At each iteration a fresh `x` is captured and all closures of line *2* reference different variables (of the same name). As the previous transformation hoists loop variables outside the loop, all anonymous functions would now share the same `x`.

In JavaScript, locally declared variables are visible within the whole function body as if they had been declared at the beginning of the function. The declaration of a new variable within the `while` body would hence deliver the same result.

```
 1: var x = 1;
 2: while (true) {
 3:   var storage = new Object();
 4:   storage.x = x;
 5:   store(function(storage_) {
 6:           return function() {
 7:             return storage_.x;
 8:           };
 9:         }(storage));
10:   x = storage.x + 1;
11: }
```

```
var x = 1;
while (true) {
  var storage = new Object();
  storage.x = x;
  with(storage) {
    var tmp_fun =
      function() { return x; };
  }
  store(tmp_fun);
  x = storage.x + 1;
}
```

FIGURE 3: Explicit closure allocation with anonymous functions on the left and `with` on the right.

SCM2JS solves the problem by pushing a new frame on the call stack (thus creating an artificial scope). In JavaScript this can be accomplished by either invoking a function, or by pushing an object onto the stack (using the JavaScript `with` statement). Both techniques are implemented in SCM2JS (they can be selected with a compiler flag). Figure 3 shows the result of both approaches. An object is allocated in line *3*, which will hold the captured variables. In line *5* the storage object is pushed onto the stack. In the first case an anonymous function is executed. The parameter `storage` is thereby copied and the capturing function (created in line *6*) hence captures a local `storage_`-object. In the second case JavaScript's `with` statement pushes the `storage` object on the stack itself. The fields contained within `storage` consequently become local variables for the enclosed statements. The capturing function saves the stack during its creation and hence holds a reference to the pushed object.
In both approaches the use of `x` in line *7* references now a different `x` for each generated function.
The impact on the performance is largely dependent on the source-program and the target browser. A very short loop, like in our example, is the worst case and is respectively 17 (`with` technique) and 28 (anonymous function technique) times slower under Firefox (than the same version without explicit closure handling). Under Opera and Konqueror the impact is less noticeable (about 3 times slower for `with` and 4-6 times slower with anonymous functions). However, this programming pattern is rare enough not to impact the performance of most programs (and none of our benchmarks).

4 TAIL CALLS

It is well known that tail calls [3] can be implemented without stack consumption when the execution platform supports `goto`. In the example of Figure 4 the calls at line *3*, and *7* are both tail calls and could be implemented with a `goto` if compiled to assembly.

```
1: function even(x) {
2:   if (x === 0) return true;
3:   else return odd(x-1);
4: }
5: function odd(x) {
6:   if (x === 0) return false;
7:   else return even(x-1);
8: }
9: is2even = even(2);
```

FIGURE 4: A simple tail call intensive program.

In languages without `goto`, such as JavaScript, most[5] tail calls can be transformed into `while` loops (as in Section 3). Our example shows that this is not always possible and there exist two other popular techniques to achieve proper tail recursion for the remaining tail calls. In the rest of this section we shortly present Baker's technique [2] and a naive version of *trampolines* [17]. Section 4.1 discusses a more efficient version of trampolines developed for the Funnel compiler [15] and our modification to make this technique compatible with native JavaScript calls. Section 4.2 then presents our tail call optimization.

Baker's technique requires the program to be transformed into Continuation Passing Style (henceforth CPS) first. Function invocations allocate frames on the stack which are used as the first generation of a generational garbage collector. Whenever the stack reaches the stack limit a garbage collection is performed, and the program restarts with an empty stack.

Trampolines on the other hand avoid tail calls by passing the target of tail calls to the caller waiting for the result of the currently running function. It is then the caller's task to invoke the received function (which itself could return another trampoline closure). The following code presents a trampoline version of the previous `even/odd` example. The code for the omitted `odd` function would be similar to the `even` function.

[5] Except for `even/odd` and `ewal` all other benchmarks were tail-call free after the `while` transformation.

```
1: function even(x) {
2:   if (x === 0) return true;
3:   else return new Trampoline(odd, x-1);
4: }
5: res_or_tramp = even(2);
6: while (res_or_tramp instanceof Trampoline)
7:   res_or_tramp = res_or_tramp.restart();
8: is2even = res_or_tramp;
```

At each tail call a new closure is allocated and returned (line *3*). The caller in line *5* then needs to restart potential trampoline-closures. In this basic form trampolines are expensive. Each tail call needs to create a closure and non tail calls have to test for trampoline closures within a `while` loop.

4.1 Efficient trampolines

A more efficient version of trampolines has been proposed by the authors of the Funnel-to-Java compiler [15]. They trade space for speed: instead of returning after each tail call, a constant number `c` of consecutive tail calls are allowed. Once this limit is reached an exception is thrown or a trampoline closure is repeatedly returned. After the `c` frames have been popped the trampoline closure is invoked. If the limit is not reached (either the function returns or reaches a non tail call), then the execution continues normally without removing the frames. Setting `c` to 1 is hence equivalent to the naive trampoline technique. A higher value yields faster programs, but consumes more memory. According to their experiments a value of about 40 seemed to be a good compromise.
SCM2JS's tail call handling resembles this technique in that it allows more than one consecutive tail call. Our implementation differs in the way the call counter is passed to functions. When the counter is passed as supplementary parameter it breaks the call convention, and interfacing with existing code becomes difficult. The naive use of global variables has its problems too: library functions do not modify the global variables, and instrumented functions would wrongly ignore them. Suppose for instance the following code where `tail_f` is an instrumented function that might throw a tail-exception and `lib_f` is a library function that comes from an existing JavaScript library:

```
1: function tail_f() {
2:   /* instrumentation and other code*/
3:   return lib_f(tail_f); // tail call
4: }
5:
6: function lib_f(f) {
7:   f(); // non-tail call
8:   remaining_code;
9: }
```

`lib_f` does not modify the global variables, and `tail_f` has hence no idea of the existence of the remaining continuation on the stack (the `remaining_code`

of `lib_f`). When `tail_f` reaches the imposed limit `c` it will throw a tail exception. `lib_f` however does not know how to handle this exception and will simply ignore it. The continuation of `lib_f` is lost.
Measures to overcome this problem are expensive and we finally decided to send the counter like a hidden additional parameter as part of the `this` object. JavaScript does not make any distinction between functions and methods. Any function can be used as method (as in `obj.f()`) or as a function (`f()`). In the first case the function `f` is a member of the object `obj` and executed as method. In the latter case `f` is simply invoked as function. Whenever a function is invoked as method, the keyword `this` points to the object as part of which it was executed (in our example `obj`). If a function is executed as simple function (and not method), then `this` points to the *global object* which contains all global variables.
Generally the `this` object is unused in functions that are not invoked as methods. SCM2JS therefore uses it to as a container for the counter value. The call target is stored as field in a unique object `TAIL_OBJECT` and then executed as method call. The field `calls` of `TAIL_OBJECT` represents the tail-call counter `c`.

```
 1: function even(x) {
 2:   var sc_tailCalls = TAIL_OBJECT.calls;
 3:   // nonTailCall();
 4:   if (x === 0) return true;
 5:   else {
 6:     if (this === TAIL_OBJECT) {
 7:        if (sc_tailCalls == MAX_TAIL_CALLS) {
 8:          return new Trampoline(odd, [x-1]);
 9:        } else {
10:          TAIL_OBJECT.calls = sc_tailCalls + 1;
11:          TAIL_OBJECT.f = odd;
12:          return TAIL_OBJECT.f(x-1);
13:        }
14:     } else {
15:       TAIL_OBJECT.calls = 1;
16:       TAIL_OBJECT.f = odd;
17:       var sc_tailTmp = TAIL_OBJECT.f(x-1);
18:       if (sc_tailTmp instanceof Trampoline)
19:         return sc_tailTmp.restart();
20:       else
21:         return sc_tailTmp;
22:     }
23:   }
24: }
25: is2even = even(2);
```

FIGURE 5: SCM2JS's optimized implementation of trampolines.

The (simplified) code in Figure 5 presents our technique on the transformed version of the previous example (without the `odd` function which would be simi-

lar to the `even` function). At the beginning of the function the counter variable `sc_tailCalls` is initialized with the tail call counter stored in the tail object.
For each tail call the function first tests if it was called as tail call (line *6*). If the test succeeds another test (line *7*) determines if `MAX_TAIL_CALLS` (our `c`) consecutive tail calls have been executed. If the limit has been reached a trampoline has to be returned (line *8*). If the limit has not yet been reached then the counter is incremented (line *10*), and the target is called as method (line *12*). No type check is necessary as the result would be returned verbatim indifferently of its type. If the procedure was not called as target of a tail call (line *14*), then it resets the counter to 1 and handles potential trampoline closures. The result of the tail call (line *17*) is tested (line *18*), and according to the result either restarted or simply returned. The `restart` method of the trampoline is responsible for restarting any potential further trampoline closures.
Note that the non tail calls (like the one in line *25*) are not modified, and that tail calls (line *12*, and *17*) are compatible with all JavaScript functions that do not access `this`. Functions that use the `this` object are methods and usually attached to some object. As method calls have a different syntax they are not instrumented by our tail call implementation.

4.2 Acyclic trampoline optimization

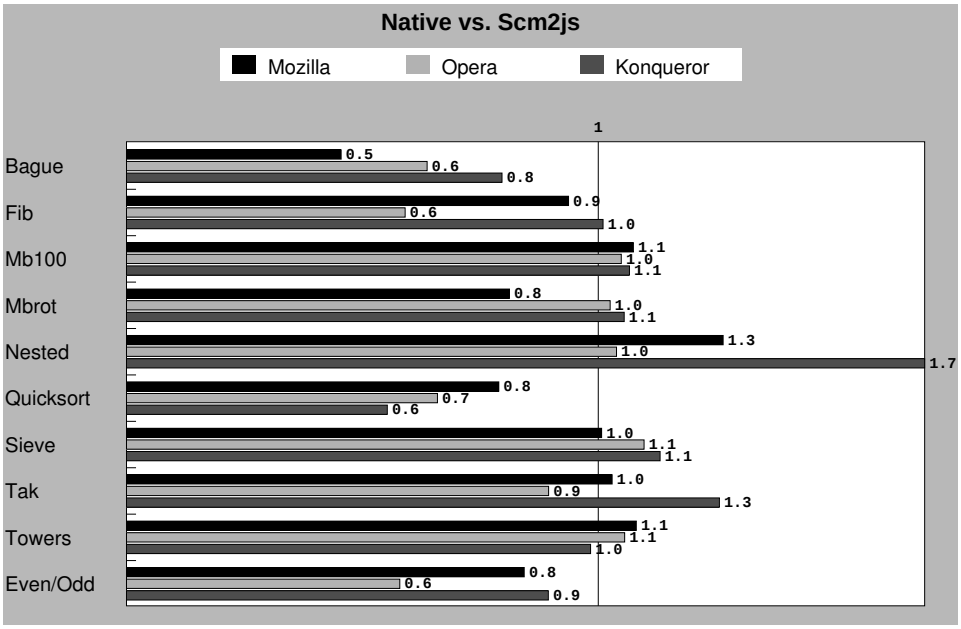
We have developed a static analysis that detects tail-call chains and potential cycles (or more importantly their absence) in them. Chains that do not finish in a cycle are compiled to direct calls without trampoline instrumentation. As such they do not test against the limit `c` anymore and may exceed the `c` consecutive tail calls. As the chain does not end in any cycle the number of supplementary calls is however bound by the length of this chain. The following code illustrates the idea:

```
1: (define (my-print msg) (print msg) msg)
2: (define (approx-print val)
3:   (if (< val 10)
4:       (my-print "small")
5:       (my-print "big")))
6: (define (len-print l)
7:   (approx-print (length l)))
```

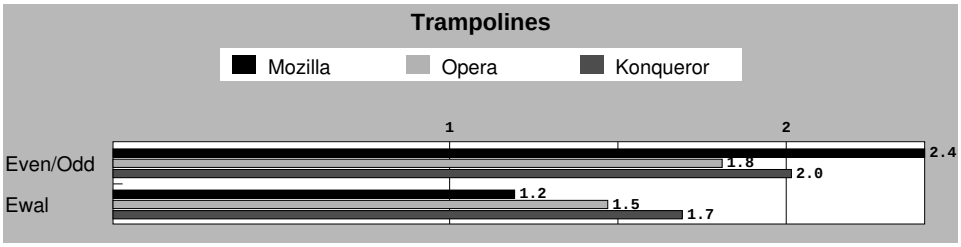
In this example there are three tail call sites (line *4*, *5*, and *7*). Furthermore the tail call chain `len-print`-¿`approx-print`-¿`my-print` does not end in a cycle. All three call locations are hence not instrumented.
If `len-print` is the c^{th} consecutive call in a tail-call chain then it should return a trampoline, but without the instrumentation it continues tail calling, thus exceeding the limit. The “damage” is however limited as there is only one other tail call afterwards. In the worst case the program hence exceeds the given limit `c` by 2 (the length of the chain).
As this optimization is done at compile time it is not possible to determine all call targets, and some tail calls keep the trampoline instrumentation even though they

can not reach any cycle. In our tail call intensive benchmark 40% of all tail calls have been simplified by this optimization.
Our experiments show that the cost for proper tail recursion is largely program-dependent. Most tail calls are loops (which are already handled by the `while` transformation) and programs tend to have few remaining tail calls. More than 80% of our benchmarks were tail-call free after the `while` transformation. Typical tail-call intensive programs however suggest a slow down of about 1.5, and extreme cases (like the `even/odd` example) run at most 2.5 times slower.

5 BENCHMARKS



(a) Compiled Scheme relative to handwritten JavaScript files, which are the 1.0 mark. Lower is better.



(b) Trampolined code relative to compiled code with the trampoline flag disabled, which are the 1.0 mark. Lower is better.

FIGURE 6: SCM2JS compiled code interpreted by Firefox, Opera and Konqueror.

To evaluate the performance of SCM2JS and trampolines we ran several bench-

marks under three Internet browsers:

- Firefox 2.0.0.2,
- Opera 9.10 build 521, and
- Konqueror 3.5.6

All benchmarks were run on an Intel Pentium 4 3.40GHz, 1GB, running Linux 2.6.20. 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.
Our first test measured the performance of SCM2JS generated code compared to handwritten JavaScript code. We wrote our benchmarks both in JavaScript and Scheme and then compared the execution time of the JavaScript version with the time of the compiled Scheme version. Figure 6a presents the ratio of the JavaScript time by the execution time of the compiled program. A value of 1.0 represents the reference time of the handwritten JavaScript code. Any value lower (resp. higher) than 1.0 means that the compiled Scheme code ran faster (resp. slower) than this code. SCM2JS fares quite well in this comparison. The compiled code generally approaches the reference value of 1.0. The good performance in `bague`, `fib`, `quicksort` and `even/odd` can be explained by our inlining pass. The bad value in `Nested` by the nature of this benchmark. `Nested` consists (as the name suggests) of several nested loops incrementing a counter in the most nested loop. The `while` loops themselves are minimal and any additional expression slows down the program. The JavaScript version `while(e--){...}` is up to 1.7 times faster than the generated version `while(e>0){... --e;}`.
Figure 6b shows the performance penalties introduced by trampolines. As SCM2JS is able to prove that none of the previous benchmarks but `even/odd` contains any cyclic tail calls (at least after the `while` transformation), enabling or disabling proper tail-recursion has no effect on the generated code. Their performance would have been equal to 1, and we therefore do not print their results. We added another benchmark (`ewal`), which implements a meta circular Scheme interpreter that executes an iterative version of `fact`. The program uses many anonymous functions and tail calls and is hence a good candidate for this test. The extreme case `even/odd` is at most 2.5 times slower and has 2 times more lines than without trampolines. The more realistic `ewal` is only about 1.7 times slower and 1.6 times bigger.

6 RELATED WORK

Related work can be classified into three categories: projects that run Scheme in Web-browsers, projects that use JavaScript as compilation target and projects that propose to unify client and server development.

There are many attempts to run Scheme and Lisp like languages on the client side. Contrary to SCM2JS these projects are either interpreters or they change the semantics of the input-language to match the semantics of JavaScript. For instance, ParenScript [9] (a compiler of a Lisp like language to JavaScript) keeps the distinction between statements and expressions from JavaScript. As such the `do` construct (compiled to JavaScript's `while` statement) can not be used at an expression location, and it does not return any value. Examples for interpreters are jsScheme [1] and Little Scheme [4].

JavaScript is a high-level language and hence not well suited as a compilation target. However, due to the ubiquity of JavaScript, such compilations have become more and more attractive.

Google [7] compiles Java to JavaScript. Java's object model can be simulated with JavaScript's prototype object model, and both share many common constructs (with identical syntax). Java is statically typed and permits many optimizations that are infeasible in highly dynamic languages like JavaScript and Scheme. The compilation from Java to JavaScript hence seems to be a good choice for efficient code. Powerful features like higher order functions and variable argument functions are however lost in the process. Due to the different nature of Java and JavaScript it is necessary to use the JSNI (JavaScript Native Interface) to interface with existing JavaScript code.

Script# [14] and NeoSwiff [6] both compile C# to JavaScript and face hence the same difficulties and share the same advantages as the Google Java compiler.

All these compilers greatly simplify the development of Web projects, but still separate client and server development. In particular the communication between client and server is still complicated.

Links [5] eliminates this boundary. Links (a typed language) uses annotations to force the execution of functions on either the server or the client, but allows the execution of non-annotated functions on either side. When calls pass the client-server boundary they are transparently compiled to xml-http-requests. The client-side portion of a program written in Links is transformed to a CPS JavaScript, which breaks the call-convention with standard JavaScript functions.

7 DOWNLOAD

SCM2JS, the compiler presented in this paper, can be downloaded at `http://-/www-sop.inria.fr/mimosa/personnel/Florian.Loitsch/scheme2js/`. It is also distributed along with Hop which can be found at `http://hop.inria.fr`.

8 CONCLUSION

In this paper we have presented SCM2JS, a Scheme to JavaScript compiler. Our work shows that such a compiler is feasible and can be efficient. We discussed the compilation of proper tail calls, one of the major differences between the two

languages. The `while` transformation we presented compiles a large percentage of tail recursive calls into cheap `while` iterations, and the trampoline implementation takes care of the rest. At the same time strict compatibility with existing JavaScript code is preserved. It is thus possible to interface easily with existing JavaScript libraries. Also SCM2JS generates efficient code. We therefore achieved both of our initial requirements for this compiler: good integration with JavaScript and good performance. The integration of SCM2JS into Hop (the framework which motivated the creation of SCM2JS) opened the door for a single language for Web programming. As Hop itself is a variant of the Scheme language it is now possible to write client-code and server-code of sophisticated web applications exclusively in Scheme.

9 REFERENCES

[1] Alex Yakovlev – **jsScheme** – *http://alex.ability.ru/scheme.html*.

[2] Baker, H. – **CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A ¡1¿** – Notices, 30(9), Sep, 1995, pp. 17–20.

[3] Clinger, W. – **Proper Tail Recursion and Space Efficiency** – Conference on Programming Language Design and Implementation, Jun, 1998.

[4] Douglas Crockford – **Little Scheme** – *http://www.crockford.com/javascript/scheme.html*.

[5] Ezra Cooper and Sam Lindley and Philip Wadler and Jeremy Yallop – **Links: Web Programming Without Tiers** – *http://groups.inf.ed.ac.uk/links/papers/links-icfp06/links-icfp06.pdf*, 2006.

[6] GlobFX Technologies – **NeoSwiff** – *http://www.globfx.com/products/neoswiff/*.

[7] Google Inc. – **Google Web Toolkit** – *http://code.google.com/webtoolkit/*.

[8] IEEE Std 1178-1990 – **IEEE Standard for the Scheme Programming Language** – *Institute of Electrical and Electronic Engineers, Inc.*, New York, NY, 1991.

[9] Manuel Odendahl and Edward Marco Baringer – **ParenScript** – *http://parenscript.org*.

[10] Manuel Serrano – **Inline expansion: when and how** – Int. Symp. on Programming Languages, Implementations, Logics, and Programs, Southampton, UK, Sep, 1997, pp. 143–147.

[11] Manuel Serrano and Erick Gallesio and Florian Loitsch – **Hop, a Language for Programming the Web 2.0** – Dynamic Languages Symposium, Oregan, USA , Oct, 2006.

[12] Manuel Serrano and Marc Feeley – **Storage Use Analysis and its Applications** – 1fst Int'l Conf. on Functional Programming, Philadelphia, Penn, USA, May, 1996, pp. 50–61.

[13] Muchnick, S. – **Advanced Compiler Design Implementation** – *Morgan Kaufmann*, 1997.

[14] Nikhil Kothari – **Script#** – *http://projects.nikhilk.net/Projects/ScriptSharp.aspx*.

[15] Schinz, M. and Odersky, M. – **Tail call elimination of the Java Virtual Machine** – Proceedings of Babel, Florence, Italy, Sep, 2001.

[16] Serrano, Manuel – **The HOP Development Kit** – Invited paper of the Seventh sigplan Workshop on Scheme and Functional Programming, Portland, Oregon, USA, Sep, 2006.

[17] Tarditi, D. and Acharya, A. and Lee, P. – **No assembly required: Compiling Standard ML to C** – ACM Letters on Programming Languages and Systems, 2(1), 1992, pp. 161–177.