Chapter 1

Hop Client-Side Compilation

Florian Loitsch[1], Manuel Serrano[1]

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. SCM2JS, the compiler we have designed, supports the whole Scheme core language. In particular, it features proper tail recursion. However complete proper tail recursion may slow down the generated code. Despite an optimization which eliminates up to 40% of instrumentation for tail call intensive benchmarks, worst case programs were more than two times slower. As a result Hop only uses a weaker form of tail-call optimization which simplifies recursive tail-calls to while-loops.
The techniques presented in this paper can be applied to most strict functional languages such as ML and Lisp.
SCM2JS 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`.

[1]Inria Sophia Antipolis, 2004 route des Lucioles - BP93, F-06902 Sophia Antipolis, Cedex, France; Email: {florian.loitsch, manuel.serrano}@inria.fr

1.1 INTRODUCTION

Hop [10] 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 [8]) 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 server code, another one specifies client code. Compiling a Hop program involves two different compilation processes. The server code is compiled to native code by a compiler that has already been described in various papers [9, 7]. The client code is compiled to JavaScript which is the imposed language for programming graphical user interfaces on the Web. Indeed, JavaScript is the only language that is supported by all major Web-browsers. Hop is an extension to the Scheme programming language [4] and this compilation is therefore mostly equivalent to a Scheme-to-JavaScript translation.

1.1.1 Main Contributions

From a practical point of view the main contribution of this work is the creation of SCM2JS, a fully functional efficient Scheme-to-JavaScript compiler. According to our observations, the Scheme compliant version with proper tail-recursion generates code that is at most 2.5 times slower than hand-written JavaScript code, but with incomplete (but usually sufficient) support for tail-recursion the compiled code is on par with hand-written code. The latter compilation mode is used in Hop and 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 1.5.1). We also propose an optimization to the tail recursion mechanism that allowed us to remove 40% of the tail call instrumentation in some benchmarks (Section 1.5.2).

1.1.2 Organization of the paper

We start by giving an overview of Hop in Section 1.2. Section 1.3 then shows how SCM2JS compiles Scheme's core language to JavaScript. In Section 1.4 we discuss function compilation. This specifically includes our `while` transformation for recursive loop functions. This transformation always improves performance. The compilation of the remaining tail calls is presented in Section 1.5. 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 1.6 shows the results of our benchmarks. Related work is discussed in Section 1.7. We finally conclude

this paper in Section 1.8.
Our compiler supports first-class continuations, but their compilation is too complex and extensive to fit into this paper and will be the subject of another publication.

1.2 HOP

In this section we briefly present the Hop programming language by an example (Section 1.2.1). A more thorough description and a discussion of its virtues compared to other Web-programming languages can be found in [10]. Once Hop has been introduced, we can then enumerate the main characteristics of the client code compilation (Section 1.2.2).

1.2.1 Hop at a glance

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 it elaborates a response which is sent to the client. That response is usually made of 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.
The code snippet in Figure 1.1 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)" w)))))
5:  (<HTML> (<INPUT> :onkeyup
6:               ~(innerHTML-set! $def ($svc this.value)))
7:          "The definitions are:" def))
```

FIGURE 1.1: Google suggest written in Hop.

When this program is invoked the server will start creating an HTML-page as response. This page (starting at line *5*) contains a `<DIV>` area (bound to the variable `def`) and a text field, which will react on `onkeyup`-events. During this elaboration stage the callback functions is compiled to JavaScript, and the complete page is then sent to the client. There, a change to the input-field triggers the callback, which updates the `def`-element. The update happens in two steps.

First the service[2] `svc` of line *2* is called with the input-field's value (accessible through `this.value`), and, as second step, the visible text of `def` is replaced by the result returned by the server (`(innerHTML-set! $def ...)`). The function `svc` executes a database query to find all words that with the given prefix. Note that (except for the database query) both server and client are written in (extended) Scheme, and that switching from one to the other can be done using only one character. Client code is introduced by a `~` (tilde) and one can escape back to server-code using `$`. This construct strongly resembles Scheme's *quasiquotes* in that `$` escaped expressions are already evaluated during construction before sending the page to the client. During that elaboration stage the reference to `def` is transformed to JavaScript code retrieving the `div`, and the service is transformed into a server call. The example in Figure 1.2 further demonstrates this property.

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

FIGURE 1.2: The service-call will not change the transmitted x-value.

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 *4* modifies the variable `x` the program will alert `0`. During elaboration of the site, `(alert $x)` had already been replaced by `(alert 0)` and the modification in the service is not transmitted to the client anymore.
One should note that while server code and client code are expressed in the same language they are intended for different purposes. The server code can access all resources of the server computer. In particular, it can access the file system, the network interfaces, or it can execute long lasting CPU intensive computations. However, it is not knowledgeable of any characteristics of the graphical user interface that are only known to the client code. The client code, on the other hand, knows everything about the graphical user interface but, for security reasons, has no access to other resources. This dichotomy between server code and client code is reflected by two different APIs that are available to the server and to the client.
In conclusion: *(i)* Hop is a functional language built on top of the Scheme programming language with which it shares most of its syntax. *(ii)* Server code and client code are expressed in the same language. *(iii)* The tilde sign `~` introduces client code and the dollar sign `$` inside client code escapes back to server code. *(iv)* A *service* is a function defined on the server (Figure 1.1, line *2*) that can be invoked from the client (Figure 1.1, line *6*). *(v)* Finally service invocations in-

[2]The `service` form creates a function that can be invoked by both client and server code, but executes always on the server.

volve transmitting and receiving complex values that can be any compound data structure.

1.2.2 Compiling Hop client code: the SCM2JS compiler

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

FIGURE 1.3: Hop program example.

We have developed a compiler, named SCM2JS, which was needed to compile Hop client code to JavaScript. Hop server code is compiled by another compiler and in Figure 1.3 only the expressions starting with `~` in line *3* and line *4* 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 [4] 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 before the compilation takes place. The example however demonstrates some additional difficulties: SCM2JS has to deal with server 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 dynamically bound variables (like `alert`).
When compiling Hop client code SCM2JS allows unbound variables, and both symbol-related difficulties are hence avoided. Server objects are straightforward to implement and, these requirements dealt with, 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, under similar conditions, Firefox executed our benchmarks nearly ten times faster than Safari. Safari is nevertheless a popular browser which tends to demonstrate that most users are not paying much attention to performance. Developers, on the other hand, are more concerned with performance, and 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 languages. Function calls should always have the same syntax, independently of where the targets are.

1.3 CORE COMPILATION

This section introduces the compilation of the Scheme core language. Function compilation and proper tail call handling are discussed in Sections 1.4 and 1.5.
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 (in particular closures) 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 expects 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[3] and the demand for optimizations require the construction of a true abstract syntax tree.
JavaScript and Scheme do not share the same data types, either. JavaScript, for instance, does not have any list data type, so SCM2JS 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[4]) are semantically compatible with their respective counterparts in JavaScript. 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

[3] A variable declaration inside a function is valid for the whole function. This is true, even, when the declaration is after the first use. As a consequence it is not possible to create cheap scopes inside functions.

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

JavaScript over a truly Scheme-conformant string implementation. 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.[5]

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 [5]. On the other hand, the optimizations that are still applicable can have a big impact on performance. For instance, our inlining pass (modeled after [7]) 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.

1.4 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. They abort after a predefined maximum function call depth and SCM2JS hence 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 1.5) limits the call stack size for the remaining tail calls.

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

In Scheme nearly all loops are implemented as recursive tail calls. Figure 1.4a demonstrates an example program with a common loop pattern. Parts enclosed into < and > are not important for our discussion and may represent any valid Scheme expression.

```
(let loop ((x 0)
           (y 0))
 (if <test>
     <body1>
     (begin
       <body2>
       (loop
        (* x 2)
        (+ x 1)))))
```

(a) a common Scheme loop pattern.

```
var x = 0, y = 0;
while (true) {
  if (<test>) {
    <body1>;
  } else {
    <body2>;
    var tmp = x;
    x = x * 2;
    y = tmp + 1;
    continue;
  }
  break;
}
```

(b) unoptimized

```
var x = 0, y = 0;
while (!<test>) {
  <body2>;
  y = x + 1;
  x = x * 2;
}
<body1>;
```

(c) optimized

FIGURE 1.4: Unoptimized and optimized `while` compilation of a common Scheme loop pattern.

Whenever SCM2JS encounters a tail call to the surrounding function it compiles this pattern into a `while` loop as in figure 1.4b and c. Note that the optimized version reorders the loop-variable assignment to avoid the temporary variable (which is compliant to Scheme's specification).
Such naive source-to-source translations are only sufficient as long as loop variables are not captured or not mutated. As the transformation reuses loop variables during each iteration explicit closure handling becomes necessary. For instance, in the following code the variable `x` is captured by an anonymous function at each iteration:

```
(let loop ((x 1))
  (store! (lambda () x))
  (set! x (* x 2))
  (loop (+ x 1)))
```

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.
SCM2JS solves the problem by pushing a new frame on the call stack (thus creat-

```
 1: var x = 1;
 2: while (true) {
 3:  var stor = new Object();
 4:  stor.x = x;
 5:  store(function(stor_) {
 6:          return function() {
 7:            return stor_.x;
 8:          };
 9:        }(stor));
10:   stor.x *= 2;
11:   x = stor.x + 1;
12: }
```

```
var x = 1;
while (true) {
 var stor = new Object();
 stor.x = x;
 with(stor) {
  var tmp_fun =
   function() {return x;};
  x *= 2;
  store(tmp_fun);
 }
 x = stor.x + 1;
}
```

FIGURE 1.5: Explicit closure allocation with anonymous functions on the left and `with` on the right.

ing an artificial scope). 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 1.5 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 `stor` is thereby copied and the capturing function (created in line *6*) hence captures a local `stor_`-object. In the second case JavaScript's `with` statement pushes the `stor` object on the stack itself. The fields contained within `stor` 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. Some artificial benchmarks (executed under the same conditions as our other benchmarks in Section 1.6) revealed that under Firefox 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 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).

1.5 TAIL CALLS

It is well known that tail calls [2] can be implemented without stack consumption when the execution platform supports `goto`. In the example of Figure 1.6 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 1.6: A simple tail call intensive program.

In languages without `goto`, such as JavaScript, most[6] tail calls can be transformed into `while` loops (as in Section 1.4). 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. The first technique, due to Henri Baker [1], 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. CPS however is expensive in JavaScript and we therefore used the second technique, *trampolines*[11], in our compiler.
In the rest of this section we briefly present a naive version of trampolines. Section 1.5.1 discusses a more efficient version of trampolines developed for the Funnel compiler [6] and our modification to make this technique compatible with native JavaScript calls. Section 1.5.2 then presents our tail call optimization.
Trampolines 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 code in Figure 1.7 presents a trampoline version of the previous `even/odd` example. The omitted `odd` function would be similar to the `even` function.
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.

1.5.1 Efficient trampolines

A more efficient version of trampolines has been proposed by the authors of the Funnel-to-Java compiler [6]. They trade space for speed: instead of returning

[6]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;
```

FIGURE 1.7: Even/Odd with trampolines.

after each tail call, a constant number `c` of consecutive tail calls are allowed. Once this limit is reached a special exception (which we will call "tail-exception") containing a trampoline for the not-yet executed call is returned. After the `c` frames have been popped the counter is reset to zero and 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 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. Take for instance the code in Figure 1.8, where `lib_f` is a library function that comes from an existing JavaScript library and `tail_f` is an instrumented function that might throw a tail-exception.

```
1: function lib_f(f) {
2:   f(); // non-tail call
3:   remaining_code;
4: }
5: function tail_f() {
6:   /* tail-calls other tail-calling functions   */
7:   /* and will eventually reach the tail-limit. */
8: }
9: lib_f(tail_f);
```

FIGURE 1.8: Library-function calling a tail-calling function.

`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`). `tail_f` will tail-call another tail-calling function, and execution will eventually reach the imposed limit c. At this moment a tail-exception is thrown. `lib_f` however does not know how to handle this exception and will simply ignore it. The continuation of `lib_f` is lost.

SCM2JS has adopted a solution to this problem that relies on JavaScript's method invocation protocol. 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 can use it as a container for the counter value. The call target is stored as a field in a unique object `TAIL_OBJECT` and then executed as a method call. The field `calls` of `TAIL_OBJECT` represents the tail-call counter c.

The (simplified) code in Figure 1.9 presents our technique on the transformed version of the previous example. At the beginning of the function the counter variable `sc_tailCalls` is initialized with the tail call counter stored in the tail object. The important data of the tail-object is thus saved, and the tail-object is free to be reused for other tail-calls.

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.

As the tail-object is reused for every tail-call, we must put tail-calls into A-Normal form [3]. Otherwise another function might change the counter-value of the tail-object after line *10* or line *15*.

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. Method calls, however, are never instrumented (not even in tail position) and are hence free to use the `this` variable.

```
 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 1.9: SCM2JS's optimized implementation of trampolines.

1.5.2 Acyclic trampoline optimization

Chain-calls 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 bounded by the length of this chain. Figure 1.10 illustrates the idea. 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. We have developed a static analysis that detects tail-call chains and potential cycles in them. When the analysis proves the absence of cycles, functions are not instrumented. Applied to the example of Figure 1.10, it successfully eliminates all instrumentation for the given functions.
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).

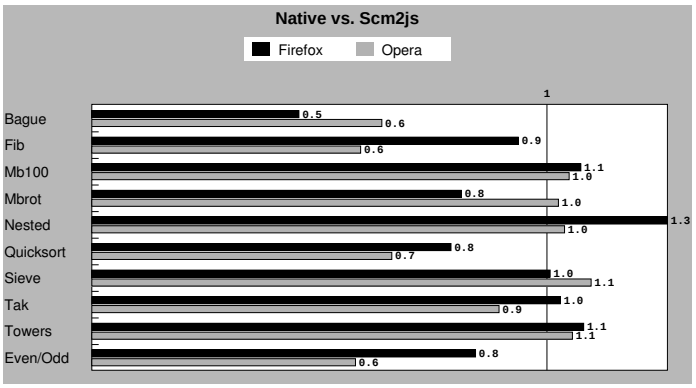
```
1: (define (my-print msg) (print msg) msg)
2: (define (approx-print val)
3:   (if (< val 10)
4:       (my-print "small")
5:       (my-print "big")))
6: (define (len-print l)
7:   (approx-print (length l)))
```

FIGURE 1.10: Chain of tail-calls reaching a non-tail-call.

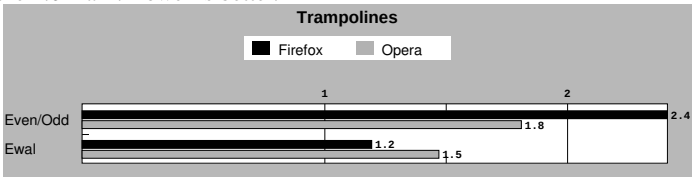
As this optimization is done at compile time it is not possible to determine all call targets, and some tail calls keep the trampoline instrumentation even though they can not reach any cycle. In our tail call intensive benchmark 40% of all tail calls have been simplified by this optimization.
Our experiments 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.

1.6 BENCHMARKS

To evaluate the performance of SCM2JS and trampolines we ran several benchmarks under two Internet browsers: Firefox 2.0.0.2 and Opera 9.10 build 521. 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 1.11a 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;}`.



(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 1.11: SCM2JS code interpreted by Firefox and Opera.

Figure 1.11b 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. The more realistic `ewal` is only about 1.7 times slower.

Although we think that code size is usually insignificant compared to the size of images that are sent with web-pages, we compared the size of the produced code with hand-written JavaScript and the original Scheme code. Most web-browsers accept compressed JavaScript files, and Figure 1.12 therefore only contains a comparison of gzipped files. As we do not have a JavaScript version of the `ewal` benchmark we used the original Scheme code as reference. Our results show that even without trampolines SCM2JS produced code is usually bigger than the equivalent JavaScript code. This can be explained by inlining and other optimizations. The trampoline versions of `even/odd` and `ewal` were respectively 2 and 1.3

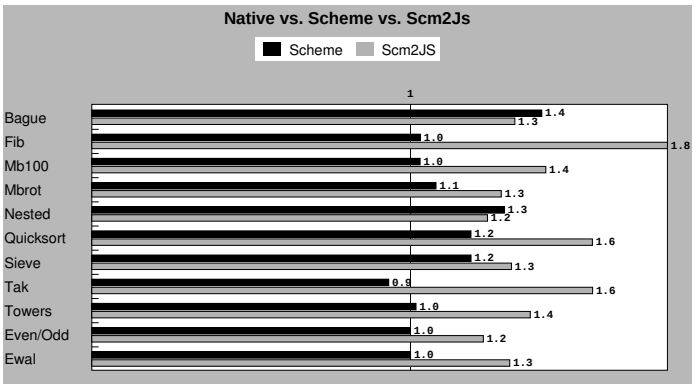


FIGURE 1.12: Code size comparison of handwritten JavaScript code, Scheme source, and SCM2JS produced code. All files have been compressed. JavaScript file size are the 1.0 mark. Lower is better.

times bigger than the version without trampolines.

1.7 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[7] (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[8] and Little Scheme[9].

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[10] 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 effi-

[7] Manuel Odendahl and Edward Marco Baringer, `http://parenscript.org`

[8] Alex Yakovlev, `http://alex.ability.ru/scheme.html`

[9] Douglas Crockford, `http://www.crockford.com/javascript/scheme.html`

[10] `http://code.google.com/webtoolkit/`

cient 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# [11] and NeoSwiff [12] 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[13] 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. It is not yet optimized for speed and runs one to two orders of magnitude slower than SCM2JS.

1.8 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 (8 out of our 10 benchmarks were tail-call free after this optimization), and the trampoline implementation takes care of the rest. Proper tail-recursion is expensive though, and even though our optimization removed the costly instrumentation for up to 40% of affected functions worst case examples exhibited a slowdown of a factor two.
We modified existing tail-call techniques so that 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.

[11] Nikhil Kothari, `http://porjects.nikhilk.net/Projects/ScriptSharp.aspx`
[12] GloxFX Technologies, `http://www.globfx.com/products/neoswiff/`
[13] Ezra Cooper, Sam Lindley, Philip Wadler and Jeremy Yallop,
`http://groups.inf.ed.ac.uk/links/papers/links-icfp06/links-icfp06.pdf`

REFERENCES

[1] H. Baker. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A <1>. SIGPLAN *Notices*, 30(9):17–20, September 1995.

[2] W. Clinger. Proper Tail Recursion and Space Efficiency. In *PLDI '98*, June 1998.

[3] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93*, volume 28(6), pages 237–247. ACM Press, New York, June 1993.

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

[5] S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[6] M. Schinz and M. Odersky. Tail call elimination of the Java Virtual Machine. In *Proceedings of Babel*, Florence, Italy, September 2001.

[7] Manuel Serrano. Inline expansion: *when* and *how*? In *PLILP '97*, pages 143–147, Southampton, UK, September 1997.

[8] Manuel Serrano. The HOP Development Kit. In *Invited paper of the Seventh* ACM SIGPLAN *Workshop on Scheme and Functional Programming*, Portland, Oregon, USA, September 2006.

[9] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *ICFP '96*, pages 50–61, Philadelphia, Penn, USA, May 1996.

[10] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Dynamic Languages Symposium*, Oregan, USA, October 2006.

[11] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. *ACM LOPLAS*, 2(1):161–177, 1992.