Compiling Scheme to 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

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

ABSTRACT

This paper presents SCM2JS a compiler that translates a variant of the Scheme programming language into JavaScript. On the one hand, some Scheme features are missing, amongst which the most important are the lack of support for continuations, the absence of exact numbers, and a partial treatment of tail recursions. On the other hand, some extensions are added for improving the connection between Scheme and JavaScript. In particular, SCM2JS extends Scheme with the JavaScript *dot-notation* which enables compact class accesses. Scheme code and JavaScript can be mixed because they both access functions and variables of the other language and because they share, at runtime, a common memory.
The codes produced by SCM2JS are fast because for most programs they have performance comparable to equivalent hand-written JavaScript programs. Hence, one may use SCM2JS for replacing JavaScript with Scheme. For instance, one may implement web libraries or HTML actions in Scheme. The paper shows how this can be done in context of Hop, a programming language dedicated to interactive web development.

1. Introduction

JavaScript is a popular scripting language. It is embedded in many applications such as PDF viewers, integrated development environments, graphical applications. It has given birth to dialects that are also successful (for instance, ActionScript, the language for programming Flash applications). Its large deployment is mainly due to its use as scripting language for web pages. Nearly every modern site uses JavaScript now, and all mainstream internet browsers are capable of interpreting JavaScript. As Web browsers are installed on nearly every computer, JavaScript interpreters are ubiquitous.
Contrary to what its name suggests, JavaScript is a functional language whose design has been influenced by the Scheme programming language [10]. However, these two languages are separated by their syntaxes, the Scheme support for continuations, the JavaScript support of object layer based on prototypes, and some other minor technical differences. Since this is not an incredibly difficult task, we have found it appealing to craft a compiler from Scheme to JavaScript for replacing the latter with the former, in particular, in active web pages.

However, for this replacement to be effective two requirements must be fulfilled.

- Using Scheme instead of JavaScript should impose no performance penalty. That is the performance of compiled Scheme codes must be comparable to equivalent hand-written JavaScript codes. We are considering performance as a potential issue even if 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 on the ultimate generation of Intel processors is about ten times faster than running the same programs within Safari. This tends to demonstrate that most users are not paying much attention to performance.
- Scheme and JavaScript must be tightly integrated. That is all bindings must be available from one language to the other. The data structure must also be usable indifferently in both language. For web programming this is of prime importance. In JavaScript, the DOM (the *Document Object model* is a standardized way of representing a visualized HTML document using an object hierarchy) is interfaced with classes. For enabling easy access to the DOM, Scheme programs must be able to access JavaScript classes with a lightweight and intuitive syntax.

For enabling the SCM2JS compiler to meet these two requirements we have decided to give up on full Scheme compliance. Section 2 presents the difference of the source language of SCM2JS and the official Scheme language as defined by its standard [10]. Section 3 discusses the Scheme to JavaScript compilation per se. It shows how Scheme data structures are mapped to JavaScript. It presents the most significant aspect of the compilation of the control flow. Then it concludes with a performance study. The following Section 4 presents the integration of Scheme and JavaScript. An example of embedding of SCM2JS is then provided in Section 5 which presents how it is used in the context of the Hop programming language. The remaining sections discuss related and future work.

2. SCM2JS and R5RS

SCM2JS is not conform to the R^{5}RS. There are mainly three reasons for the non-conformance of SCM2JS: extensions to the Scheme language, short-comings (mainly motivated by efficiency reasons) and missing features.
SCM2JS should integrate easily with JavaScript. During the creation of a comfortable interface we were led to extend the Scheme language by the “dot-notation” as explained in Section 4.
R^{5}RS requires compliant Scheme implementations to be properly tail-recursive. SCM2JS correctly translates some common recursive function calls into iterative statements (see Section 3.3), but fails to cover all cases. Instead of implementing some expensive

techniques, like trampolines, we decided to leave the remaining tail-recursive calls untouched. Efficiency made us violate another R5RS requirement. JavaScript does not have any fixed point number type, and instead of reimplementing a integer type we decided to map Scheme's exact numbers to JavaScript's floating point numbers.
SCM2JS is not (yet) feature complete either. It misses hygienic macros (although `define-macro` can be used as replacement) and it does not implement the complete Scheme runtime library: most importantly `call/cc` and `eval` are not available. `eval`, on one hand, could be easily implemented in the form of a library without any changes to the compiler. `call/cc`, on the other hand, would require either a transformation to Continuation Passing Style, or exception handling mechanisms as described in [14]. Both techniques would induce a certain overhead on the compiled program.

3. SCM2JS Compiler

JavaScript and Scheme are related languages, but the compilation from one to another is not so trivial. In [9] we already presented a JavaScript to Scheme compiler. This section introduces the inverse compiler, SCM2JS, and the important points of such a compilation. We will begin with a short comparison of the two languages, then continue with the chosen data type mapping in Section 3.2. We will discuss flow control compilation in Section 3.3. Section 3.4 presents some optimizations we implemented in SCM2JS, and Section 3.5 shows some benchmarks which confirm that all these efforts weren't fruitless.

3.1 Scheme vs. JavaScript

This section discusses the differences (and similarities) of JavaScript and Scheme. We limit this comparison to the most significant parts of the two languages. JavaScript has been inspired by Scheme, and most of the similarities are hence not astounding. The following list enumerates some of the shared features:

- types are dynamically checked,
- functions are first class citizens,
- lexical scoping,
- automatic memory management,
- an *eval* function, which allows one to compile and run code at run-time, and
- n-ary and var-arg functions with an `apply` primitive that allows to indirectly call these functions on a list or array of arguments.

In other areas JavaScript is however quite different from Scheme. The move from Lisp-style syntax to C-like syntax is certainly the most striking difference, and Section 3.4 mentions the expression to statement pass which has been implemented as a consequence of the C-like syntax. Other changes have nevertheless far greater impact on the development of a Scheme to JavaScript compiler. Especially JavaScript's peculiar syntactic scoping of variables makes the compilation difficult. In JavaScript a variable declaration is valid for a complete function block wherever the declaration is located. Section 3.3 shows why this behavior complicates the compilation.
As we will see in the next section, the data types aren't equal either. JavaScript has less data types, and the matching types are not always equivalent. JavaScript strings, for instance, are (contrary to Scheme strings) immutable, and JavaScript numbers are always floating point (whereas Scheme has exact numbers too).
Another difference is the lack of `call/cc` in JavaScript. The existing `try/catch` partially compensates for the absence, but is not as powerful as `call/cc`

3.2 Data Types

JavaScript has basically four main types: booleans, numbers, strings, and objects. Functions are of type object and don't have their proper data type. The JavaScript specification [8] additionally cites `undefined` and `null` as respective types. Scheme, despite being a smaller language, uses four more types: pairs, vectors, characters and symbols.
SCM2JS maps Scheme's

- booleans to booleans.
- functions to JavaScript functions.
- numbers to JavaScript numbers. Even though this mapping seems obvious, it makes SCM2JS non conform to R5RS. JavaScript's numbers are always floating point, whereas Scheme differentiates exact and inexact numbers.
- pairs to the JavaScript "class" `sc_Pair` with fields `car` and `cdr`. The empty list is represented by `null`.
- vectors to JavaScript `Array`s.[1]
- characters to the "class" `sc_Char`, holding a JavaScript string of length 1.
- strings to a new JavaScript "class", `sc_String`. It is not possible to reuse JavaScript's strings, as they are immutable. `sc_String` itself holds one of these immutable strings, and replaces it, when necessary.
- symbols to strings.

JavaScript's typing rules are less restrictive than Scheme's rules. Due to implicit conversions many operations that are errors in Scheme are valid expressions in JavaScript. SCM2JS takes advantage of R5RS' liberal error handling. Implementations are rarely forced to detect and signal errors, but are free to handle most errors the way they want to. In particular "[...] it is an error for a procedure to be passed an argument that the procedure is not explicitly specified to handle [...]. Implementations may extend a procedure's domain of definition to include such arguments". In our case, errors are handled by JavaScript. If an operation yields an error according to the JavaScript specification, an error is signaled. If however JavaScript is able to handle an expression that would be invalid in Scheme, no error is reported. The following examples demonstrate this behavior: `(car '())` is an invalid expression in Scheme. SCM2JS compiles this expression to `null.car`. As `null` doesn't have any fields, JavaScript throws an exception. The compiled version hence raises an error. In the following (invalid) code snippet we try to add a symbol to a number: `(+ 3 'sym)`. The translated code would be `3 + 'sym'` which is a valid JavaScript expression concatenating the two strings `"3"` and `"sym"`. In this case no error will be detected.
Similar missed errors happen when functions are not passed the correct number of arguments, in which case either missing arguments are filled with a special `#unspecified` value, or additional arguments are ignored.

3.3 Flow Control

Scheme has only few flow-control constructs, and all of them have similar counterparts in JavaScript. A direct mapping is however impossible. R5RS requires Scheme implementations to be properly tail-recursive. JavaScript on the other hand doesn't even mention this feature. Mapping Scheme function calls naively to JavaScript function calls is hence dependent on the JavaScript implementation

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

and not always[2] conform. Two popular solutions are *trampolines* and Henry Baker's "Cheney on the M.T.A." [6]. The concept of trampolines needs the modification of all call locations. Tail calls are replaced by instructions that save the call target and arguments in global variables, followed by a `return`. Non tail calls, on the other hand, are wrapped into an iterative loop. Initially the original call target is executed. Once the function returns the trampoline checks for the existence of a function in the global variable that temporarily holds the tail call targets. If this variable is not empty the stored function is executed, and the trampoline waits for the next return.

Even though trampolines solve the initial problem of rapidly growing stacks, they are inefficient. Tail calls need to manipulate the global variables, whereas non tail call must be wrapped into an iterative loops. An optimized version of trampolines is presented in [11]. Instead of returning at every tail call, the program is allowed to stack a constant number of tail calls. Only when the limit is reached the continuation is stored in the global variables, and the trampoline is used.

"Cheney on the M.T.A.", on the other hand, transforms the program into Continuation Passing Style (CPS) first, and thereby eliminates the need for stacks. In the paper the authors propose to use the stack as short term memory. Whenever the stack reaches the stack limit a garbage collection is performed, and the program restarts with an empty stack. The stack becomes the youngest generation in a generational garbage collector.

All these techniques introduce some performance penalty and we therefore decided not to implement any. For now SCM2JS just tries to compile tail-recursive calls into iterative JavaScript statements. This approach is however incomplete as it is impossible to statically determine all call-targets. SCM2JS is able to transform the most common recursive calls (like `let loop` constructs) but leaves other more difficult tail recursive calls untouched.

Even when the call-targets have been determined the tail-rec to iterative transformation is not trivial. A `let loop` construct, for instance, is mapped to JavaScript's `while` statement.

Example:

```
(let loop ((x 1))
  (if (> x 10)
      'done
      (loop (+ x 1))))
```

is compiled as:

```
var res = undefined;
var x = 1;
while (true) {
  if (> x 10)
     res = "done";
  else {
    x = x + 1;
    continue;
  }
  break;
}
```

This simple mapping is efficient but it may break down when closures are built in the body of the loop:

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

[2] Actually all important JavaScript interpreters are known **not** to be tail-recursive.

In this example the loop variable `x` is captured by anonymous functions in line *2*. The captured `x` is however freshly allocated at each recursive call and is hence different for each of these functions. An invocation of two different closures would yield two different results. The previous transformation on the other hand hoists loop variables outside the loop, which implies a shared `x` for all anonymous functions:

```
1: var res = undefined;
2: var x = 1;
3: while (true) {
4:   store(function() { return x; });
5:   x = x + 1;
6:   continue;
7: }
```

`x` is now outside the loop (line *2*) and it's value is physically modified at each iteration (line *5*). As the variable is allocated outside the loop all anonymous functions reference the same `x`. As this variable is physically changed during each iteration, all anonymous functions return the same result (i.e. the final value of `x`) when executed.

In JavaScript locally declared variables are visible within the whole function body, and declaring a new variable within the `while` body doesn't solve this issue. The following code demonstrates this unfruitful attempt:

```
var res = undefined;
var x = 1;
while (true) {
  var tmp_x = x;
  store(function() { return tmp_x; });
  x = x + 1;
  continue;
  break;
}
```

Even though `tmp_x` is declared inside the `while` construct, the peculiar JavaScript scoping rule makes it visible with the complete function. In practice this means that the order and location of variable declarations is ignored and that there exists never more than one variable of the same name. Once a variable has been declared future declarations of variables of the same name are ignored. In our case this means that the anonymous functions share the same variable (this time `tmp_x`) again.

There are only two ways of creating new scopes in JavaScript: functions and the `with` construct. Indeed anonymous functions solve the captured variable problem:

```
var res = undefined;
var x = 1;
while (true) {
  (function(x) {
    store(function() { return x; });
  })(x);
  x = x + 1;
  continue;
  break;
}
```

Although they partially defeat the purpose of the iterative `while` statement (i.e. removing the unnecessary applications), they don't fill the stack and are hence a great improvement over recursive function calls. It is furthermore possible to limit the anonymous function to parts of the loop-body. If the captured variable is only used within one branch of an `if` it is not necessary to invoke the function in the other branch. Anonymous functions are however too limiting: some JavaScript constructs don't work over function

boundaries, and in particular the `continue` statement must not be moved inside another function.
JavaScript's `with` statement on the other hand fullfills all our requirements. `with` takes an object as parameter and pushes it on the execution context stack. Every field of the pushed object becomes a local variable limited to the `with` scope. It is hence sufficient to create an empty object, store the captured variables in the fields of this object, and finally use the `with` construct to push it on the execution context stack.

```
var res = undefined;
var x = 1;
while (true) {
  var tmp = new Object;
  tmp.x = x;
  with(tmp) {
    store(function() { return x; });
  }
  x = x + 1;
  continue;
  break;
}
```

Some benchmarks indicate that, depending on the interpreter, pushing an empty object on the execution context stack is slightly slower or faster than function calls.

3.4 Optimizations

This section discusses common optimizations and compiler techniques and shows how they apply to our compiler SCM2JS. Due to the high level of JavaScript only few optimizations are applicable. It is for instance not easy to take advantage of a typing pass as one can't pass typing information to JavaScript interpreters. We did however implement (amongst others) an inlining optimization.
SCM2JS's inlining is done in two steps. A first pass inlines user functions, whereas a second pass inlines runtime procedures. The first pass is rather rudimentary but fulfills its design goal, to inline all `let loop` expressions. When a variable is bound to a function and when it is only used once in functional position, then it is inlined. The first condition avoids us to do a control flow analysis, whereas the second condition avoids code growth.[3] Even though this optimization is quite limited, it reduces the execution time of some benchmarks to less than 50%
A straightforward compilation of Scheme programs to JavaScript programs maps Scheme calls to JavaScript calls. In many cases this translation introduces an overhead. In particular binary operations like `+`, `-`,or `%` are far less efficient if called via a runtime function. A second pass therefore replaces function calls to specific library functions with the more efficient version: `sc_plus(x, y)` would become `(x + y)`. Other examples for this optimizations are all list primitives (`car`, `cons`, `null?`, etc.) and vector functions (`vector-ref`, `vector-set!` or `vector-length`). This optimization is especially important as it gives a speed improvement of up to a factor of 25.
We conclude this section with a problem many Scheme compiler encounter: Scheme is expression based, whereas JavaScript is statement based. Some JavaScript constructs can only be used at specific locations. It is for instance not possible to use a `while` loop at the right hand side of an assignment. SCM2JS introduces temporary variables to store the intermediate results of such statements. Interestingly these statements are mostly the result of optimizations. JavaScript has an equivalent expression for most statements: the if statement `if (test) s1 else s2` can be replaced by the conditional operator `test ? e1 : e2` and the block statement `{ s1 s2 }` by the sequence expression (`e1, e2`). The tail-rec pass however introduces some statement-only `while` loops and makes the technique necessary. Some other JavaScript statements without equivalent expressions are `continue`, `return`, `break`, `throw`, and `try`.

[3] Future versions will probably inline too if the function's size is under a certain constant.

3.5 Benchmarks

In order to evaluate the performance of SCM2JS we ran several benchmarks under three Internet browsers on three different architectures:

- Linux/x86: an Intel Pentium 4 3.40GHz, 1GB, running Linux 2.6.15.

  The used browsers were:

  - Firefox 1.5.0.1,
  - Opera 9.0 pre2, and
  - Konqueror 3.5.1

- Apple/G4: a PowerPC G4 1.67GHz, 2GB, running Mac OS X 10.4.5.

  We used the following browsers:

  - Firefox 1.5.0.1,
  - Opera 9 build 3303, and
  - Safari 2.0.3 (417.9.2)

- Apple/Core: a Intel Core Duo 2GHz, 1GB, running Mac OS X 10.4.5.

  We used the following browsers:

  - Firefox pre 1.5.0.2,
  - Opera 9 build 3278, and
  - Safari 2.0.3 (417.9.2)

It turned out, that the choice of browser has far more impact on the performance than the architecture. `Firefox` and `Opera` are sometimes up to ten times faster than `Safari` or `Konqueror`. The fastest architecture Intel Core Duo is however only four times as fast as the PowerPC G4. More importantly, the browsers behave similar on the different platforms.
Each program was run 10 times, and the minimum time was collected. The time measurement was done by a small script, which launched the benchmarks. Any time spent on the preparation (parsing, precompiling, etc.) was hence not measured.
Every benchmark has been written in Scheme and JavaScript. We then compiled the Scheme versions using SCM2JS and measured the execution times of both the JavaScript code and the compiled version.
Figure 1, 2 and 3 present the ratio of the JavaScript time by the execution time of the compiled program (resp. on the Pentium 4, Apple G4 and Apple Intel Core Duo). A value of 1.0 therefore 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.
In general the compiled code is nearly as fast as the handcrafted version. Even under the worst condition SCM2JS is only about six times slower than the JavaScript version (benchmark `Nested` under Firefox on the Pentium 4 in figure 1). `Nested` consists of several nested loops incrementing a counter in the most nested loop. SCM2JS introduces a temporary variable for each loop, and uses this variable twice per iteration. This more than doubles the size of the loop bodies with a respective performance penalty. Apparently `Opera` handles this case much better than `Firefox`. `Firefox` is about three times faster than `Opera` when run on the JavaScript files, but `Opera` degrades more gracefully and beats `Firefox` on the compiled files.

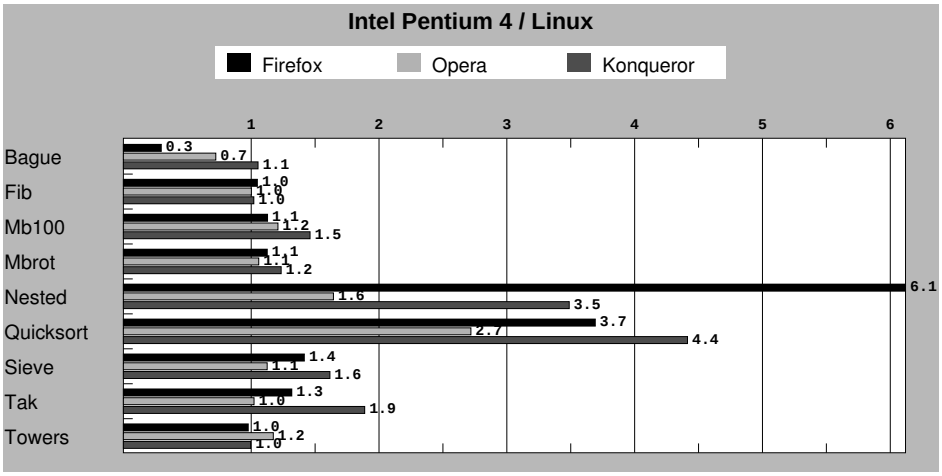

Figure 1. SCM2JS compiled Scheme code interpreted by Firefox, Opera and Konqueror on a Pentium 4 running Linux. Scores are relative to handwritten JavaScript files, which are the 1.0 mark. Lower is better.

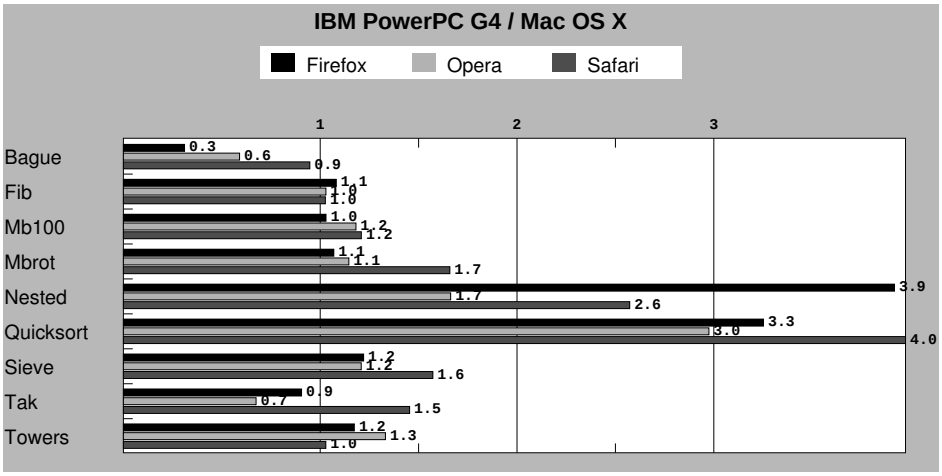

Figure 2. SCM2JS interpreted by Firefox, Opera and Safari vs. JavaScript on a Apple G4 running Mac OS X. Lower is better.

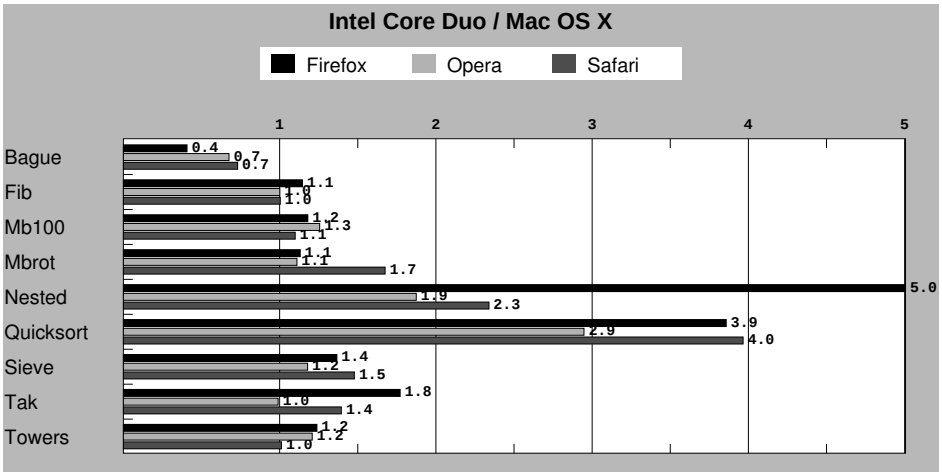

Figure 3. SCM2JS interpreted by Firefox, Opera and Safari vs. JavaScript on a Apple Intel Core Duo running Mac OS X. Lower is better.

The Quicksort benchmark suffers from the same problem: a very small loop iterates over an array to find elements smaller or greater than a certain pivot.
In less extreme cases, SCM2JS performs quite well, though, and the rudimentary inlining even allows SCM2JS to beat the handwritten JavaScript in some cases (in particular the Bague benchmark).

4. JavaScript Integration

SCM2JS provides a complete JavaScript-Scheme interface. That is JavaScript code can access Scheme variables and call Scheme functions, and inversely Scheme code can use JavaScript values. When passing values from one language to another, conversions must take place. In particular JavaScript strings and Scheme strings are not of the same type. As JavaScript strings are immutable, it was not possible to directly map Scheme strings to them. Scheme symbols, on the other hand, are immutable too, and in the current version symbols are compiled to JavaScript strings, whereas Scheme strings are translated into a new class `sc_String`. If a Scheme string should be used as JavaScript string a call to `string->symbol` is hence necessary[4], whereas a JavaScript string can be converted to a Scheme string by the means of `symbol->string`. Pairs and characters don't have any conversion routines (as there are not any equivalent data types in JavaScript), and it is mostly convenient to just use their compiled representation. Pairs by accessing their `car` and `cdr` fields, and characters by accessing the `val` field of the `sc_Char` class.
Functions are already mostly interchangeable, but to ease the integration of Scheme functions in the context of JavaScript we added the `this`-extension. If a function is used as method, JavaScript sets a `this` variable[5] to the object on which the method was called. Take for instance the code in Figure 4.

```
1: function f() { print(this.x); }; // prints 3
2: var o = new Object;
3: o.x = 3;
4: o.f = f;
5: o.f();
```

Figure 4. JavaScript's `this` keyword represents the object on which the method has been called.

As `f` is called as method of `o`, (line *5*) `this` will be set to `o` within the function `f` (line *1*). The `this`-extension brings the `this` variable to Scheme functions. Code within a procedure may reference `this` and hence access the object's field on which it has been called. Using `this` outside the scope of methods doesn't make much sense (even though the JavaScript specification gives a sense to this case), and it should therefore be only used if a Scheme function is going to represent a method.
In order to use JavaScript objects within Scheme two runtime primitives `js-field` and `js-field-set!` have been added. They allow to respectively retrieve or set the fields of JavaScript objects. The use of these functions is not optimal though. Accessing (or changing) fields is too verbose and cumbersome. The equivalent of the short JavaScript expression

```
x.y.z = a.b
```

would be:

```
(js-field-set! (js-field x 'y) 'z (js-field a 'b))
```

[4] It is obviously possible to directly use a symbol instead.

[5] Actually `this` is not a variable, but a keyword.

We introduced therefore JavaScript's dot-notation into Scheme. In addition to `js-field-set!` and `js-field` SCM2JS makes it possible to directly access fields by concatenation the object, a dot and the field-name. The previous example becomes:

```
(set! x.y.z a.b)
```

The dot-notation leads us to the first way of referencing JavaScript variables from within Scheme: the runtime Scheme variable `*js*` holds all global variables. If one wants to access JavaScript's global `alert` function, it is sufficient to retrieve the `alert` field of the `*js*` variable: `(js-field *js* 'alert)` or using the dot-notation `*js*.alert`.
Internally `*js*` is set to JavaScript's global `this` object. As such it automatically contains all global variables.[6] The `*js*` object is also an easy way of making Scheme values available to JavaScript code. Just by assigning a new field of `*js*` users can create a new global variable and thereby export a Scheme value.[7] In addition to this explicit exports SCM2JS automatically declares all global Scheme variables as global Javascript variables. The global variables are declared under their mangled form, which means that only a subset of variables are exported under their Scheme names.
In a certain way the `*js*` variable resembles explicit casts of statically typed languages like Java or C. The programmer is explicitly telling the compiler he is going to do something dangerous, but that he is aware of the risk. Indeed, SCM2JS can't verify if the JavaScript variable exists, or if the variable name has been mistyped.
A more secure method uses the `js` keyword. Whenever SCM2JS encounters a `(js . A-LIST)` expression in the top-level it adds the a-list's bindings to the symbol-table. These symbols are resolved at compile time and accesses to these variables are therefore checked by the compiler. This approach has the additional advantage of better integration into the code. The A-list allows to alias JavaScript variables with typical Scheme symbols. In the following example `A_GLOBAL` would be aliased to `*global*` and `setEvent` to `event-set!`:

```
(js (event-set! setEvent) (*global* A_GLOBAL))
```

In some cases even the indirect access over the `*js*` variable is not dynamic enough. Scheme code snippets can be compiled separately and it is often desirable to access variables of other pieces. These snippets might be the event handlers scattered around an HTML file, or complete libraries.
The previous interfaces can both be used to bind the separated pieces, but both come with their respective disadvantages. The safe method requires the developer to write import clauses. Especially in the case of event handlers, these clauses can become cumbersome. Accessing variables of different parts over the `*js*` variable is not optimal either. Global Scheme variables become members of the global JavaScript variables under their mangled name. Even though they are hence accessible over the `*js*` variable the programmer would need to know the mangled variable name. As the mangling function should stay compiler intern we implemented a third option: every unbound variable is automatically considered to be an imported global variable. If SCM2JS needs to mangle the name the mangling will be the same for both modules, otherwise the variable access could either reference a global JavaScript variable,

[6] The `*js*` variable is, under its mangled name, a field of this object too.

[7] This approach even allows to create global variables that are only accessible through the global object: `(set! *js*.new-global 'val)` sets the global variable `new-global` to `'val`. `new-global` is however parsed as `new - global` and it is hence impossible to reference the variable without the use of the global `this` object.

a "compatible" Scheme variable. Also, this model combines some advantages of the previous approaches: imported variables don't need to be declared, but can't be used instantly, and they can be accessed directly (without redirection by the `*js*` variable). This flexibility comes at a price, though. Allowing unbound symbols removes an important safety net as any unbound symbol will be considered to be a global imported variable. That is, the compiler is no longer able to display "unbound variable" error messages. Figure 5 demonstrates this smooth integration of JavaScript into Scheme on a small example that manipulates the DOM. The given Scheme program dynamically adds new HTML elements into an existing tree.

5. Hop integration

In this section we present an example of SCM2JS embedding. We show how it can be used in the context of Hop. Hop is a functional language designed for programming webapplications [13]. It exposes a distributed model made of two *strata*. The first stratum, named the *main* stratum, is intended for programming server-side computations. The second stratum, named the *client* or *GUI* stratum, is intended for programming client-side computations. The two strata execute on different computers. They do not share memory but they share their namespace and they communicate via function calls and events.

A Hop program first starts on a server. This initial step elaborates an HTML tree that is sent to the client, typically a web browser. User interactions on the client side may lead to the invocation of functions on the server which, in turn, elaborates new HTML trees that are sent back to the client.

In the current version of Hop, the main strata is programmed in a variant of Scheme and the client strata is programmed in JavaScript. The SCM2JS compiler allows to use Scheme for both stratum. This section presents the integration of SCM2JS in Hop. It first presents a compact overview of Hop (Sections 5.1, 5.2, and 5.3). Then, it focuses on the integration of SCM2JS (Section 5.4 and Section 5.5) and its impact on the GUI stratum.

5.1 The syntaxes of Hop

Hop rests on the closeness of the syntaxes of Scheme and HTML. A simple syntactic transformation turns HTML documents into Hop programs. Hop adds an extra open parenthesis before opening markups and replaces closing markups with single closing parentheses. In addition Hop attributes are introduced by an identifier starting with a colon character (`:`) and their value is separated from the name by white spaces. Hence the HTML document of Figure 6 is written as shown Figure 7 in Hop.

```
<HTML>
   <BODY>
      <TABLE width="100%">
         <TR> <TD>0</TD></TR>
         <TR> <TD>1</TD></TR>
         <TR> <TD>2</TD></TR>
         <TR> <TD>3</TD></TR>
      </TABLE>
   </BODY>
</HTML>
```

Figure 6. A simple HTML file.

In the plain version of Hop, the JavaScript expressions of the GUI stratum are delimited by opening and closing curly braces ({ and }). For the sake of the example, Figure 8 shows a program that displays the local time of the client.

```
(<HTML>
   (<BODY>
      (<TABLE> :width "100%"
         (<TR> (<TD> 0))
         (<TR> (<TD> 1))
         (<TR> (<TD> 2))
         (<TR> (<TD> 3)))))
```

Figure 7. A simple HOP program.

```
(<HTML>
   (<BODY>
      (<P> "The current date is: ")
      (<SPAN> :id "date" "")
      (<SCRIPT> {
          var el = document.getElementById( "date" );
          el.innerHTML = new Date() + "";
      })))
```

Figure 8. Blending Scheme and JavaScript syntax.

5.2 Hop elaboration

The HTML tree that forms an answer is computed by the expressions of the main stratum. That is, it is *elaborated* on the server. During that stage, values can be *injected* inside expressions of the GUI stratum. This is denoted by the escape character `$` inside curly braces. The expression following a `$` belongs to the main stratum. It is evaluated during the elaboration. Its results is inserted in the expression of the GUI stratum. Simple atomic values such as strings or numbers as well as compound values such as vectors, references to HTML nodes, and, as presented in Section 5.3, functions can be inserted. The source code of Figure 9 illustrates this capacity. First, line *1* a HTML `span` element is declared. It is inserted in the answer line *5*. A reference to this element is injected in the expression of the GUI stratum line *8*. In this example, a second expression is injected line *9*.

```
1: (let ((el (<SPAN> :id "date" "")))
2:    (<HTML>
3:       (<BODY>
4:          (<P> "The current date is: ")
5:          a-span
6:          (<SCRIPT> {
7:              ($el).innerHTML =
8:                 "client time: " + new Date() +
9:                 " -- server time: " + $(current-date);
10:           })))))
```

Figure 9. Elaborating a HTML tree.

5.3 Hop services

The expressions of the two strata of a Hop program are evaluated in different heaps and environments. That is, they do not share data. However, they may communicate by the means of *service invocations*. A service is a function declared on the server. It is defined using the `define-service` form. As any function it may accept several arguments. A service is invoked from the GUI stratum with a special `hop` form:

```
hop( service( a0, a1, a2, ...), callback )
```

The values $a_0, a_1, a_2, \ldots$ are the actual arguments sent to the service. Once the server has completed the evaluation of the service's body, i.e., when it has completed the elaboration of the answer, it applies, on the client, the *callback* function on the service's answer (converted to a string). The code of Figure 10 shows an example

```
1:  (define (table-create! . rows)
2:     (let ((table (document.createElement 'TABLE)))
3:        (for-each (lambda (row)
4:                    (table.appendChild row))
5:                  rows)
6:        table))
7:
8:  (define (row-create! header? . cell-texts)
9:     (let ((row (document.createElement (if header? 'TH 'TR))))
10:       (for-each (lambda (cell-text)
11:                   (let ((c (document.createElement 'TD)))
12:                      (set! c.innerHTML cell-text)
13:                      (row.appendChild c)))
14:                 cell-texts)
15:       row))
16:
17: (define (div-fill! div-id)
18:    (let* ((nb-images document.images.length)
19:           (nb-forms document.forms.length)
20:           (head-row (row-create!  #t 'Tag 'Count))
21:           (image-row (row-create! #f 'Image nb-images))
22:           (form-row (row-create! #f 'Form nb-forms))
23:           (table (table-create! head-row image-row form-row)))
24:       ((document.getElementById div-id).appendChild table)))
```

Figure 5. A Scheme program manipulating the DOM tree. The given program dynamically adds an HTML table, that contains statistics (the number of images and forms) about the current document. In line *18*, and *19* the number of images and forms of the current document are retrieved. Using the `createElement` method of the `document` variable we then construct new table tags (line *2*, *9*, and *11*). These tags are subsequently combined via `appendChild` (line *4*, and *13*), and finally appended to the `div` tag that has been given as parameter (line *24*).

where a table of contents is build on the server from information sent by the client. In addition to illustrating the service call line *10*, this example also shows that compound data such as vectors may transit from client and server and vice versa.

```
1:  (define-service (make-toc sections)
2:     (<OL> (vector-map <LI> sections)))
3:
4:  (<HTML>
5:     (map <H1> (iota 3))
6:     (let ((toc (<DIV> "Toc: ")))
7:        (<BUTTON>
8:           :onclick {
9:              var hs=document.getElementByTags("h1");
10:             hop($make-toc(hs),
11:                 function(s) {($toc).innerHTML = s;})
12:          } "Click to view the table of contents")))
```

Figure 10. Service invocation.

5.4 Hop Scheme

The SCM2JS compiler lets us replace JavaScript with Scheme in Hop programs. In this new version, the curly braces are replaced with the `~` escape character that introduces expressions of the client stratum. During the elaboration stage, Scheme client expressions are compiled on the fly into JavaScript. The source code of Figure 11 is a direct re-writing of Figure 10.

As it can be noticed, the dot-notation of SCM2JS is strongly relevant in the context of Hop. It enables a compact notation for reading and writing fields of instances which are frequent in Hop.

In combination with SCM2JS, we have added a new construction to Hop, namely the `with-hop` form. Its syntax is:

```
(with-hop (service a0 a1 a2 ...) continuation)
```

```
1:  (<HTML>
2:     (map <H1> (iota 3))
3:     (let ((toc (<DIV> "Toc: ")))
4:        (<BUTTON>
5:           :onclick
6:           ~(let ((hs (document.getElementByTags "h1")))
7:               (hop ($make-toc hs)
8:                    (lambda (s)
9:                      (set! $toc.innerHTML s))))
10:          "Click to view the table of contents")))
```

Figure 11. Service invocation in Scheme.

This form invokes the *service* with the arguments a_0, a_1, a_2, ... On completion of the invocation, the continuation is applied. Contrary to the `hop` service invocation presented in Section 5.3, the value sent to the continuation is no longer a string but any simple or compound data type. In particular, Hop is able to automatically create, on demand, classes on the client. That is, when an instance is returned to the client as a result of a service invocation, in addition to sending the object, Hop also sends to the client the code required for declaring the class anteriorly. This enables Hop expressions to access objects independently of the strata. This is illustrated in the example of Figure 12. In this example, a class `software` is declared in the main stratum (line *24*). Instances are created in the service `query-by-name` and sent back to the client (line *10*). On the GUI stratum (line *27*, *28*, and *29*) the value returned by that service is directly accessed as a class instance.

5.5 External Scheme Files

The previous section detailed only one way of inserting Scheme code into Hop documents. Another way consists of including complete Scheme files (usually libraries) in the `<HEAD>` section of the page. Figure (ref :figure "file-include") shows an example where two files are included: one JavaScript file, and one Scheme

```
 1: (module example
 2:    (library sqlite)
 3:    (class software
 4:       name::string
 5:       author::string
 6:       version::string
 7:       license::string))
 8:
 9: (define-service (query-by-name name)
10:    (sqlite-select make-software "softwares WHERE (name = ~a)" name))
11:
12: (let ((name (<INPUT>))
13:       (version (<INPUT>))
14:       (license (<INPUT>)))
15:    (<HTML>
16:       (<BODY>
17:          (<TABLE>
18:             (<TR> (<TD> name))
19:             (<TR> (<TD> version))
20:             (<TR> (<TD> license)))
21:          (<INPUT>
22:             :type 'text
23:             :onkeyup ~(if (= event.keyCode 13)
24:                           (with-hop ($query-by-name this.value)
25:                              (lambda (o)
26:                                 (when o
27:                                    (set! $name.value o.name)
28:                                    (set! $version.value o.value)
29:                                    (set! $license.value o.value)))))))))
```

Figure 12. This example illustrates the exchanges of compound values from a server to a client. In this example, a class is declared on the server (line *3*). The service `query-by-name` (line *9*) queries a database in order to return instances of the class `software` to the client. Automatically, Hop declares the class `software` on the client. So, in the client code, the value received in the continuation (line *25*) is tested and the fields revelant to the application are directly extracted (line *27*, *28*, and *29*).

file. The Scheme file is compiled on the fly and then sent the same way as the JavaScript file.

```
(<HTML>
   (<HEAD>
     (<HOP-HEAD> :jscript "jslib.js")
     (<HOP-SCHEME-HEAD> :sscript "schemelib.scm"))
   (<BODY> "some body text"))
```

Figure 13. A Hop document includes a JavaScript and a Scheme file.

We mentioned in Section 4 that there are different ways of interfacing with Scheme code. Either the safe interface using `js` directives, the explicit mode accessing the `*js*` variable, or the unsafe way where all unbound symbols are considered to be implicitly imported global variables.
In the context of Hop we opted for the unsafe interface. It is the easiest way of combining external files and injected code. Even if external file came with a header file a safe compilation would still be extremely difficult. Scheme code in Hop files is often out of order (the code for a button click might be before the main script with important initializations and declarations) and all injected Scheme expressions are compiled upfront when the file is loaded. The out of order compilation implies potentially undefined variables at early locations, and the upfront compilation means that we don't have any relationship between the different Scheme pieces. Two Scheme expressions might be part of the same page (and should hence share the same variables) or might be completely separated entities (or even dead code). The unsafe interface has however advantages too, and we found it comfortable to use JavaScript libraries without the hassle of import/export declarations.

5.6 Conclusion

Replacing JavaScript with Scheme on the client stratum has its pros and cons. On the positive side, it enforces the cohesion between the two strata. Scheme and JavaScript are similar languages but they promote slightly different programming styles. For instance, Scheme promotes recursions and higher-order operators, while JavaScript promotes loops and iterators. Hence, while replacing JavaScript with Scheme does not bring new functionality to Hop, it, undoubtably, makes programs look nicer. A single, coherent syntax is used which gives the feeling of a continum from the server to the client and vice-versa. Switching from server programming to client programming does not require much intellectual effort.
On the dark side, it could be argued that using a single syntax, inside a single source code, for expressions evaluated in different environments and libraries is prone to error. The early experiments tend to show that is not a dramatic drawback. Only the experience will tell if using a single syntax for both strata is a benefit.

6. Related Work

Several different projects have tried to replace JavaScript on the client side. The attempts can be divided into three categories: either by enhancing the Internet browsers with their language of choice, by compiling to already existing plugins (in particular the Java Virtual Machine), or by compiling directly to JavaScript.
The most famous and successful extensions to browsers are the Java and the Flash plugin. The Java extension [2] integrates a JVM into the browser, and allows to run arbitrary[8] code within it. Macromedia's Flash plugin [1] on the other hand is specialized in displaying animations. Since release 5 Flash comes with an

[8] The interpreted code is contained by security measures. But the engine itself is capable of executing any code.

integrated scripting language ActionScript [7], which is based on ECMAScript.
A more SCM2JS related project is OpenScheme [4]. Similar to SCM2JS it allows to use Scheme as scripting language on the client side. It achieves this through the means of a plugin (available for either Internet Explorer or Netscape compatible browsers) that is capable of interpreting Scheme code. With this plugin developers can directly send Scheme code to the client.
Most projects don't write their own plugins, though, but reuse the existing Java extension. Any language that can be compiled to Java Byte Code can be interpreted by the Java plugin, and is hence executable by the browser. Bigloo [12] amongst many others can benefit from this approach to deliver so called applets with web-pages.
The Java plugin is however not installed on every client, and a JavaScript based approach reaches a far bigger user base. `lisp2JavaScript` [3] is a prototype of compiling lisp-like languages to JavaScript. `ParenScript` [5] is an already more mature attempt with a complete macro environment or lisp-style iterations. The language itself is largely based on JavaScript (with a LISP syntax), which makes a direct translation straight-forward.

7. Future Work

SCM2JS has reached a usable state and it can be used as JavaScript replacement now. There is however still room for improvement and we would like to address the following issues in the future:

- increase performance. We think that an improved version of our inlining pass, and some peephole optimizations could have a positive impact on performance.
- improve the runtime. Some functions like `eval` are still missing, even though their implementation is straightforward.
- implement `call/cc`. We hope to be able to create serializable continuations, which would allow us to research migration between Web browsers.

8. Conclusion

In this paper we have presented SCM2JS, a Scheme to JavaScript compiler. We enumerated some differences between these two languages, and showed how they affect an efficient compilation. We detailed in particular the tail recursive loop translation. Several benchmarks demonstrate the usability of our compiler. SCM2JS compiled Scheme code is generally not more than twice as slow as handcrafted JavaScript code.
We introduced the JavaScript and Hop integration. On the client side the JavaScript interface allows to exchange data between Scheme and JavaScript code, and an extension eases the use of JavaScript values within Scheme code. On the server-side we are able to replace JavaScript completely with Scheme.
As Hop itself is a variant of the Scheme language it is now possible to write sophisticated web applications exclusively in Scheme.

9. References

[1] **http://www.macromedia.com/shockwave**.

[2] **www.java.com/getjava/**.

[3] **http://www.cryptopunk.com/wip/lisptojavascript.html**.

[4] **http://gd.tuwien.ac.at/languages/OpenScheme/nposm.htm**.

[5] **http://blogs.bl0rg.net/netzstaub/archives/000525.html**.

[6] 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.

[7] Colin Moock – **ActionScript for Flash MX: The Definitive Guide** – , , 2002, pp. 1104 (est.).

[8] ECMA – **ECMA-262: ECMAScript Language Specification** – 1999.

[9] Florian Loitsch – **Javascript to Scheme Compilation** – 2005 Workshop on Scheme and Functional Programming, September, 2005.

[10] Kelsey, R. and Clinger, W. and Rees, J. – **The Revised(5) Report on the Algorithmic Language Scheme** – Higher-Order and Symbolic Computation, 11(1), Sep, 1998.

[11] Schinz, M. and Odersky, M. – **Tail call elimination of the Java Virtual Machine** – Proceedings of Babel&aps01, Florence, Italy, Sep, 2001.

[12] Serpette, B. and Serrano, M. – **Compiling Scheme to JVM bytecode: a performance study** – 7th Int&apsl Conf. on Functional Programming, Pittsburgh, Pensylvanie, USA, Oct, 2002.

[13] Serrano, M. and Gallesio, E. – **HOP, a language for programming the Web** – 2006.

[14] Tatsurou Sekiguchi and Takahiro Sakamoto and Akinori Yonezawa – **Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling** – Lecture Notes in Computer Science, 20222001, pp. 217+.