Javascript to Scheme Compilation

Florian Loitsch

Inria Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France

Florian.Loitsch@sophia.inria.fr

ABSTRACT

This paper presents Jsigloo, a Bigloo frontend compiling Javascript to Scheme. Javascript and Scheme share many features: both are dynamically typed, they feature closures and allow for functions as first class citizens. Despite their similarities it is not always easy to map Javascript constructs to efficient Scheme code, and in this paper we discuss the non-obvious transformations that needed special attention.

Even though optimizations were supposed to be done by Bigloo the chosen Javascript-Scheme mapping made several analyses ineffective and some optimizations are hence implemented in Jsigloo. We illustrate the opportunities Bigloo missed and show how the additional optimizations improve the situation.

1. Introduction

Javascript is one of the most popular scripting languages available today. It was introduced with Netscape Navigator 2.0 in 1995, and has since been implemented in every other dominant web-browser. As of today nearly every computer is able to execute Ecmascript (Javascript's official name since its standardization [9] in 1997), and most sophisticated web-sites use Javascript.

Over the time Javascript has been included in and adapted to many different projects (eg. Qt Script for Applications, Macromedia's Actionscript), and it is not exclusively used for webpages anymore. Most of them are interpreting Javascript, but some are already compiling Javascript directly to JVM byte code (eg. Mozilla's Rhino [5] and Caucho Resin [4]).

Javascript is not easy to compile though. Several of its properties make it challenging to generate efficient code:

- Javascript is dynamically typed,
- functions are first class citizens,
- variables can be captured by functions (closures),
- it provides automatic memory management, and
- it contains an *eval* function, which allows one to compile and run code at run-time.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

Scheme has similar features, and Scheme compilers are faced with the same problems. Contrary to Javascript much research has been spent in compiling Scheme, and there exists several efficient Scheme compilers now. By compiling Javascript to Scheme it should hence be possible to benefit from the already present optimizations. Bigloo, one of these efficient compilers, has the supplementary advantage of compiling to different targets: in addition to C, it is capable of producing JVM bytecode or .NET's CLI. A Javascript to Scheme compiler would hence immediately make it possible to run (and interface) Javascript with these three platforms.

When we started the compiler we expected to have the following advantages over other Javascript compilers:

- The compiler should be small. Most of Javascript's features exist already in Scheme, and only few adjustments are needed.
- The compiler should be easy to maintain. A small compiler is easier to maintain than a big, complex compiler.
- The compiler should be fast. Bigloo is fast, and if the translated code can be optimized by Bigloo, the combined compiler should produce fast code. An efficient Javascript to Scheme compiler does not need to create efficient Scheme-code, but code that is easily optimized by Bigloo.
- Any improvement in Bigloo automatically improves the Javascript compiler. New optimizations are automatically applied to the Javascript code, and new backends allow distribution to different platforms.
- Javascript code could be easily interfaced with Scheme and all languages with which Bigloo interfaces.

Many existing Javascript compilers or interpreters already featured some of the listed points, but none combined all these advantages.

Our compiler, *Jsigloo*, takes Javascript code as input, and translates it to Scheme code with Bigloo extensions [1] which is then optimized and compiled to one of the three platforms. Furthermore it is planned to integrate Jsigloo into Bigloo (as has been done for Camloo [16]) thereby eliminating the intermediate Scheme-file.

Section 2 will detail the differences between Javascript and Scheme. In Section 3, the chosen data-structure mapping and typing issues are discussed. Section 4 describes the code generation and how encountered difficulties are handled. Some preliminary performance results are given in Section 5. Section 6 shows why Jsigloo is not yet finished and what needs to be improved in the future. Finally, Section 7 concludes this paper.

[1] Most of the used extensions increase Bigloo's efficiency and could be either omitted or replaced by equivalent (slower) Scheme expressions.

2. Javascript vs. Scheme

Javascript and Scheme share many features, and this section will therefore concentrate on their differences rather than similarities. Even though Javascript is generally considered to be an object oriented language, it bears more resemblance to functional languages like Scheme than to most object oriented languages. In fact Javascript's object system is based on closures which is a feature typically seen in functional languages.

Javascript's syntax resembles Java (or C), and even readers without any Javascript knowledge should be able to follow the provided code samples.

2.1 Binding of Variables

In Scheme, new variables can only be created within certain expressions (eg. `let` and `define`) which ensure that every variable is defined. Javascript however is more flexible:

- Globals do not need to be declared. They can be defined within the global scope (using the same syntax as is used for local variables in functions), but it is also possible to declare them implicitly when assigning an undeclared variable[2]. The inverse - reading from an undeclared variable - is not possible and throws an exception.
- A variable declaration (`var x;`) allows one to declare variables anywhere in a function. The variable is then set to *undefined* at the beginning of the function. Most languages provide blocks to limit the visibility of variables whereas in Javascript blocks do not influence the scoping. But even more surprising the declaration also affects all previous occurrences of the same symbol. In theory one could put all variable-declarations in a block at the end of a function.

This flexibility comes at a price though. When variables share the same name it is easy to accidently reference the wrong variables and produce buggy code. The following example contains several common mistakes.

```
1:  var x = "global";     // global variable
2:  function f() {
3:    x = "local";        // references local x
4:    var someBool = true;
5:    var x = 2;
6:    some_bool = false;  // oops.
7:    if (someBool) {
8:      var x = 1;        // references same x
9:    }
10:   return x;
11: }
12: f();       // => 1
13: x;         // => "global"
14: some_bool; // => false
```

Due to the local declaration of `x` in line *5* **and** *8* the assignment in line *3* does not change the global `x`, but the local one. Line *6* contains another annoying bug: instead of changing the local `someBool` a new global `some_bool` is created and set to `false`.

From a compiler's point of view these differences are mostly negligible though. Only the automatic assignment of *undefined* is of concern, as it makes typing less efficient.

2.2 Object System

Whereas Bigloo uses a CLOS-like [6] object system, Javascript adopted a prototype based system [13]: conceptually objects are ordinary hash tables with an attached prototype field. Whenever a property (Javascript's synonym for "member") is read (`obj.property` or `obj["property"]`) the object's hash table is searched for this entry. If the hash table contains the property the value is returned otherwise the search recursively continues on the object stored in the prototype-field. Either the member is eventually found, or the prototype does not hold an object, in which case *undefined* is returned. Writing on the other hand is always done on the first object (ie. the prototype is completely ignored). If the property did not already exist it will be created during the write.

Methods are just regular functions stored within the object. Every procedure implicitly receives a `this` argument, and when called as method (`obj.method()` or `obj["method"]()`) `this` points to the object (as in line *7* of the next example). If a function is called as non-method (line *4*) the `this`-argument is set to the *global object* which represents the top-level scope (containing all global variables and functions).

```
1: function f() {
2:    print(this);
3: }
4: f();     // 'this' in f becomes the global object
5: var o = new Object();
6: o.f = f;
7: o.f();   // 'this' in f becomes o
```

In Javascript all functions are objects, and while function invocations usually do not access the contained properties, the `prototype`-property is retrieved, when functions are used as constructors. Indeed, constructors too are just functions and do not need to be declared differently. An object creation is invoked by the construct `new Fun()`, which is decomposed and executed in three steps:

- Javascript creates a new object.
- it retrieves the `prototype`-property out of the function object's hash table (which is not necessarily identical to the prototype-field of the same object), and stores the result in the prototype-field of the newly created object.
- it runs `Fun` as if it was invoked as a method on the new object, hence allowing to modify it.

Even though the previous description is not entirely complete (we intentionally omitted some special cases), it is not difficult to show that prototype-based object-systems allow most (if not all) usual Smalltalk [11] or CLOS operations. In particular inheritance, private members or mix-ins [10] are easily feasible. Interested readers are referred to [7] for a more in-depth discussion of Javascript's object-system.

2.3 Global Object

Simply spoken, the *global object* represents the scope holding all global variables (including the functions). What differentiates Javascript from many other languages is the fact, that this object is accessible to the programmer. It is hence possible to modify global variables through an object. Interpreters simply reuse their Javascript-object structure for all global variables. Whenever needed they just provide a pointer to this structure. However for an optimizing compiler the global object is a major obstacle. The following example demonstrates how the global object disallows simple optimizations like inlining.

```
1: function g() { /* do something */ }
2:
3: function f(o) {
4:   o.g = undefined;
5:   g();
6: }
```

Suppose the given functions are part of a bigger program. Function `f` is calling the global function `g`. If `g` is never changed (eg. `g`

[2] Note, there exists a third method involving the "global object".

= some_value;), which is usually easy to detect, a good compiler could inline g. In Javascript it is however more or less impossible to be sure that g is never modified. Even the object passed to f could be the global object, and f could change g. As pointer-analyses are generally very costly and compute only conservative approximations, tracking the global object is not an option.

It is not even possible to avoid the use of global objects (as should be done with the with-construct). The global object is accessed by two ways: it is assigned to the this variable in the global scope (easily avoidable), but it is also passed to every function call, where it becomes the this-variable. Exceptions are all method-calls where the global object is replaced by the object on which the method is executed (Section 2.2 shows an example).

2.4 Variable Arity Functions

Scheme and Javascript both allow variable arity functions, but their approach is quite different. Scheme procedures must explicitly allow supplementary parameters, whereas Javascript functions are automatically prepared to receive *any* number of arguments. Even if a function's signature hints several parameters, it can still be called without passing any argument. The missing values are automatically filled with *undefined*:

```
1: function f(x) { print(x); }
2: f(); // => prints "undefined"
```

If the procedure needs to know the actual number of passed arguments, it can access the arguments-object which is available within any function. Not only does the property size hold the actual number of parameters, it also contains a reference to *all* arguments: arguments[n] accesses the nth argument. Variables in the function's signature are just aliases to these entries. The following example demonstrates the use of arguments. It will print 2, 3 and finally 2:

```
1: function f(x) {
2:   print(arguments.size); // => 2
3:   x = 3; // modify first argument
4:   print(arguments[0]);   // => 3
5:   print(arguments[1]);   // => 2
6: }
7: f(1, 2);
```

2.5 Eval Function

Scheme and Javascript both have the eval function, which allows to compile and execute code at runtime. They do not use the same environment for the evaluation, though. Scheme gives the developer the choice between the *Null-environment*, *Scheme-report-environment* or the *Interaction-environment*. The *Null-environment* and *Scheme-report-environment* are completely independent of the running program and an expression evaluated in them will always yield the same result. The optional *Interaction-environment* however allows to interact with the running program. The visibility of this environment is usually restricted to the top-level of the running program, and it is certainly independent from the location where eval is executed.[3].

Javascript, on the other hand, uses the same environment in which the eval function is executed. The evaluated code has hence access to the same variables any other statement at the eval's location would have. To ease the development of Javascript compilers, the standard gives writers the choice to restrict the use of eval to the form eval(...) (disallowing for instance o.eval(...)) and to forbid the assignment of eval (making f=eval illegal)). It is then possible to statically determine all locations and environments of eval.

3. Data structures and Types

The Javascript specification defines six types: *Undefined*, *Null*, booleans, strings, numbers and Object. This section presents the chosen representation of these types in the compiled code. Javascript's strings and booleans are directly mapped to their Scheme counterparts. As reimplementation of Javascript's numbers would have been too slow and too time-consuming, numbers are mapped to Scheme doubles. This representation does not conform to the ECMA specification[4], but the differences are often negligible. *Undefined* and *Null* are both constants and currently represented by Scheme symbols. As null is generally used for undefined objects we might replace it by a constant object in future versions of Jsigloo to improve typing.

Javascript objects however could not be mapped to any primitive Scheme (or Bigloo) type. In Javascript properties can be added and removed to objects at run-time, and Bigloo's class-system does not allow such modifications. As a result a Bigloo class Js-Object has been written that represents Javascript objects. It contains a hash table as container for these dynamic properties and a prototype-field which is needed for Javascript's inheritance. Several associated procedures simulate Javascript's property accesses and Javascript's objects are now directly mapped to the Js-Object and its methods.

Javascript functions are objects with an additional field containing the Scheme procedure. In our case Js-Function is a Bigloo class deriving from Js-Object, where a new field fun holds the procedure. A function call gets hence translated into a member-retrieval (with-access) followed by the invocation of the received procedure. Figure 1 shows the two classes and the js-call-function executing the call. (a description of this-var and arguments-vec is found in Section 4.4).

```
 1: (class Js-Object
 2:    props  ; hashtable
 3:    proto) ; prototype
 4:
 5: (class Js-Function::Js-Object
 6:    fun::procedure)  ; field of type procedure
 7:
 8: (define-inline (js-call fun-obj this-var arguments-vec)
 9:    (with-access::Js-Function fun-obj (fun)
10:       (fun this-var arguments-vec)))
```

Figure 1. Javascript's objects and functions are mapped to Bigloo classes

Javascript is dynamically typed and variables can hold values of different types during their lifetime. Most of the time programmers do not mix types though, and it is usually possible to determine a small set of possible types for each variable. Bigloo already performs an efficient typing analysis [15], but it cannot differentiate Javascript types that have been mapped to the same Scheme type (*undefined* and *null* become both symbols, objects and functions are both translated to Bigloo objects). Bigloo lacks Javascript-specific knowledge too. Depending on the operands some Javascript operations may return different types. One of these operations is the +-operator. If any operand is a string the result will be a string, otherwise the expression evaluates to a number.

As a result Jsigloo contains itself a typing pass. Contrary to Bigloo Jsigloo only implements an intraprocedural analysis resembling the implementations found in "Compiler Design Implemen-

[3] The standard is rather unclear about what this environment really represents.

[4] Javascript requires -0 and +0 to be different, which is not possible with any Scheme number type in R5RS.

tation" [14], Chapter "Data-Flow Analysis". This choice implies that parameters need to be typed to *top* (i.e. an abstract value denoting any possible type) as is the case for escaping variables: at every function-call the types could change and they need to be set to *top*. Despite these two restrictions the typing pass is able to type most expressions to some small subset. As we will see Javascript does many automatic conversions, and restricting the type-set only a little helps a lot to reduce the impact of them.

4. Compilation

Similar to Bigloo Jsigloo is decomposed into several smaller passes, which respectively execute a specific task. This first part of the section will provide a small overview over Jsigloo's architecture. The remainder of the section will then focus on the code generation. The generic case is handled first, specially treated constructs are then discussed separately. Primarily Scheme-foreign constructs like `with` (Section 4.3) and `switch` (Section 4.2) are examined in their respective subsections, but the important function compilation has its own area (Section 4.4), too. Whenever a generated code is dependent on previous optimizations we will revisit the concerned passes.

A first lexing/parsing pass constructs an abstract syntax tree (AST) composed of Bigloo objects representing Javascript constructs. Bigloo uses a CLOS like object system and it is hence possible to create procedures that dispatch calls according to their type. Jsigloo does not use any other intermediate representation other than this AST. Passes just modify the tree or update the information stored in the nodes.

An early expansion pass then removes some syntactic sugar and reduces the number of used nodes. Immediately afterwards the "Symbol" pass binds all symbols to variables. The following pass continues the removal of syntactic sugar. The optimization passes and typing is then executed before Jsigloo reaches the backend.

The code generator still receives an AST and a simplified version just needs to transform recursively the nodes to Scheme expressions and definitions. Ignoring the previously mentioned special cases and some last optimizations this transformation is straight-forward. Jsigloo just recursively dumps the nodes using generic functions and methods which are dispatched according to the type of their first argument (`define-method`). Figure 2 contains the implementations of the generic method `generate-scheme` for the `Block` and `If` nodes as well as the procedure `generate-indirect-call` used for creating unoptimized function calls.

Javascript and Scheme are very similar, and this can be seen at this level: many implementations of `generate-scheme` just retrieve the members of the node (`with-access`), transform them, and plug them into escaped Scheme lists. Most of the time only minor adjustments are needed. The `If`-method at line *9*, for instance, needs to boolify the condition expression first. That is, in Javascript 0, `null`, `undefined` and the empty string are also considered to be `false`, and conditional expressions need hence to test for these values. As we already know the type (or a superset of possible types) of every expression, some of these tests can be discarded at compile time. Instead of generating adapted code for every boolify-expression Jsigloo uses macros. This way some complexity is moved outside the compiler itself into the runtime library. Macros are still evaluated at compile time, but now within Bigloo. The `js-boolify-generate-scheme` function retrieves all possible types of the given expression and passes them to the `js-boolify_typed` macro (figure 3) as second parameter (the first one being the transformed expression). The macro then automatically discards all impossible configurations[5].

Similar `_typed`-macros are used in many other places. Even though properties of Javascript objects are always referenced by strings (`obj.prop` is transformed into `obj["prop"]`), the expression within the brackets can be of any type. Javascript therefore performs an implicit conversion to string for every access. For instance the `0` in `obj[0]` is automatically converted into `"0"`. `obj[0]` and `obj["0"]` reference hence the same property. The conversion is in this case performed by the `->string_typed`-macro which reduces the tests as much as possible. Another implicit conversion is executed for numeric operators which convert their operands to numbers (`->number_typed`). Generally every conversion has its `_typed` pendant which is used whenever possible.

```
1:  (define-method (generate-scheme b::Block)
2:     (with-access::Block b (elements)
3:        `(begin
4:           #unspecified ; avoid empty begin-blocks
5:           ,@(map generate-scheme elements))))
6:
7:  (define-method (generate-scheme iff::If)
8:     (with-access::If iff (test true false)
9:        `(if ,(js-boolify-generate-scheme test)
10:           ,(generate-scheme true)
11:           ,(generate-scheme false))))
12:
13: (define (generate-indirect-call fun this-arg args)
14:    ; JS ensures left-to-right evaluation of arguments.
15:    (if (or (null? args)         ; 0 arguments
16:            (null? (cdr args))) ; 1 argument
17:        `(js-call ,fun
18:                  ,this-arg
19:                  (vector ,@(map out args)))
20:        (let ((tmp-args (map (lambda (x)
21:                               (gensym 'tmp-arg))
22:                             args)))
23:    `(let* (,@(map (lambda (tmp-name arg)
24:                     (list tmp-name
25:                           (out arg)))
26:                   tmp-args
27:                   args))
28:              (js-call ,fun
29:                       ,this-arg
30:                       (vector ,@tmp-args))))))
```

Figure 2. the `generate-scheme-code` methods for some selected nodes.

Some Javascript constructs need more than just these minor adjustments though. In particular `switch`, `with` and even the well known `while` do not have corresponding Scheme expressions. Due to various optimizations, functions too are not directly mapped to their Scheme counterparts and are therefore discussed in a separate subsection.

4.1 While Translation

The straightforward intuitive compilation of

```
1: while(test) body
```

to

```
1: (let loop ()
2:    (if test
3:        (begin
4:          body
5:          (loop))))
```

[5] The actual `js-boolify_typed` in the Jsigloo-runtime even removes the test for the type, if the expression can only have one single type. The given code sample also misses some other object-tests.

```
(define-macro (js-boolify_typed exp types)
   (let ((x (gensym 'x)))
      `(let ((,x ,exp))
          (cond
             ,@(if (member 'bool types)
                   `(((boolean? ,x) ,x))
                   '())
             ,@(if (member 'undefined types)
                   `(((eq? ,x 'undefined)
                      #f))
                   '())
             ,@(if (member 'null types)
                   `(((eq? ,x 'null)
                      #f))
                   '())
             ,@(if (member 'string types)
                   `(((string? ,x)
                      (not (string=? ,x ))))
                   '())
             ,@(if (member 'number types)
                   `(((number? ,x)
                      (not (=fl ,x 0.0))))
                   '())
             (else #t)))))
```

Figure 3. `js-boolify_typed` used the calculated types to optimize the conversion.

misses an important point: loops in Javascript can be interrupted (`break`) or shortcut (`continue`). These kind of break-outs require either `call/cc` (or similar constructs) or exceptions. Jsigloo uses Bigloo's `bind-exit`, a `call/cc` that can only be used in the dynamic extend of its form:

```
(bind-exit (break)
   (let loop ()
      (if test
          (begin
             (bind-exit (continue)
                (body))
             (loop)))))
```

In the current Bigloo version non-escaping `bind-exits` are not yet optimized though[6] and a bind-exit removal pass has been implemented.

We used `bind-exits` not just in loops, but also for the `switch`-breaks (see next section) or the function-`returns`. In certain cases there is no easy way of avoiding them, but the following transformations are able to remove most of them. The following three samples represent some cases where our analysis allows to eliminate `bind-exits`.

```
(lambda (x)
   (bind-exit (return)
    (if (eq? x 'null) (return 'undefined))
    ;do something
   ))
```

```
(bind-exit (return)
   ; do something
   (if test
       (return 'any)
       (return 'thing))
)
```

[6] Bigloo's `bind-exit` supplies a closure, which, when invoked, unwinds the execution flow to the end of `bind-exit`'s definition (not unlike exceptions caught by a `catch`). Jsigloo uses only a small part of `bind-exit`'s functionality. The supplied closure never leaves the current procedure, and in this case invocations of bind-exit can be transformed into simple `gotos`. Future versions (post 2.7) of Bigloo will contain such an optimization.

```
(lambda (x)
   (bind-exit (return)
    ; do something
    (return result)))
```

All these examples are based on the `return` statement, but similar examples exist with the `continue` keyword of the `while` statement.

Our optimization relies on two observations:

- If an `if`-branch does not finish its execution but is interrupted (`break`, `continue`, `return` or `throw`) any remaining statements following the `if` can be attached to the other branch of the `if`[7].
- Any invocation of the escaping closure, directly followed by the end of the surrounding `bind-exit` is unnecessary and can be removed.

The first observation allows to transform the first example into:

```
(lambda (x)
   (bind-exit (return)
    (if (eq? x 'null)
        (return 'undefined)
        ;do something
     )))
```

Under the assumption that the `returns` have not been used elsewhere in the code, all `bind-exits` can now be removed thanks to the second rule:

```
(lambda (x)
    (if (eq? x 'null)
        'undefined
        ;do something
     ))
```

```
; do something
 (if test
     'any
     'thing)
```

```
(lambda (x)
    ; do something
    result)
```

This optimization removes all but one `bind-exit` from the 33 `bind-exits` found in our test-cases and benchmarks.

4.2 Switch Construct

Javascript's `switch` statement allows control to branch to one of multiple choices. It resembles Scheme's `case` and `cond` expressions, which serve the same purpose. As we will see, neither of them has the same properties as the Javascript construct, and `switch` therefore need to be translated specially.

Javascript permits non-constant expression as `case` clauses and in the following example `expr1`, `expr2` and `expr3` could thus represent any Javascript expression (including function-calls):

```
switch (expr)
   case expr1: body1
   case expr2: body2
   default:    default_body
   case expr3: body3
```

It is therefore not possible to map `switch` to Scheme's `case` which only works with constants. Scheme's `cond`, on the other

[7] If neither branch finishes normally, the remaining statements are dead code, and can hence be removed.

hand, evaluates arbitrary expressions, and if it was not for Javascript's "fall-throughs", a `switch` statement would be easily compiled into an equivalent `cond` expression:

```
1: (let ((e expr))
2:   (cond
3:    ((eq? expr1 e) body1)
4:    ((eq? expr2 e) body2)
5:    ((eq? expr3 e) body3)
6:    (else default_body))
```

As it is, a case-body falls through and continues to execute the body of the next case-clause (unless, of course, it `break`s out of the switch). To simulate these fall-throughs Jsigloo wraps the bodies into a chain of procedures. Each procedure calls the following body at the end of its corps and hence continues the control-flow at the beginning of the next clause's body. `break`s are simply mapped to `bind-exit`s and are not yet specially treated.[8]

The following code demonstrates this transformation applied to our previous example:

```
 1: (bind-exit (break)
 2:  (let* ((e expr)
 3:         (cond-body3   (lambda () body3))
 4:         (cond-default (lambda ()
 5:                         default-body
 6:                         (cond-body3)))
 7:         (cond-body2   (lambda ()
 8:                         body2
 9:                         (cond-default)))
10:         (cond-body1   (lambda ()
11:                         body1
12:                         (cond-body2))))
13:   (cond
14:    ((eq? expr1 e) (cond-body1))
15:    ((eq? expr2 e) (cond-body2))
16:    ((eq? expr3 e) (cond-body3))
17:    (else (cond-default)))))
```

Even though Javascript's default clause does not need to be the last clause, it is only evaluated once all other clauses have been tested. It is therefore safe to use the `cond`'s `else`-clause to invoke the default body, but care must be taken to include its body in the correct location of the procedure-chain.

4.3 With Statement

The access to the property *prop* of a Javascript-objects `obj` is usually either done by one of the following constructs: `obj.prop` or `obj["prop"]`. A third construction, the `with`-keyword, pushes a complete object onto the scope stack which makes all contained properties equivalent to local variables. Within interpreters this operation is usually trivial. The interpreter just needs to reuse the Javascript object type as representation of a scope. When it encounters a `with` it pushes the provided object onto their internal scope-stack. Compilers do not use explicit scope objects though, and pushing objects onto the stack is just not feasible.

Moreover, an efficient compilation of the `with`-statement is extremely difficult. As Javascript is a dynamically typed language it is not (always) possible to determine the type and hence the properties of the `with`-object. Even worse: Javascript objects might grow and shrink dynamically. It is possible to add and remove members at runtime. The following code shows an example where a variable within a closure references two different variables even though the same object is used.

[8] The current transformation has been implemented following a suggestion of a reviewer, and it was not possible to remove the `bind-exit`s in this short time-frame.

```
 1: var o = new Object();
 2: function f(x)
 3: {
 4:    with(o) {
 5:       return function() { return x; };
 6:    }
 7: }
 8: g = f(0);
 9: g();     // => 0;
10: o.x = 1; // adds x to o
11: g();     // => 1;
```

During the first invocation (line *9*) of the anonymous function of line *5* the `o` object does not yet contain `x` and the referenced `x` is hence the one of the function `f`. After we added `x` to `o` another call to `g` references the object's `x` now.

It is therefore nearly impossible to find the shadowed variables when entering a `with`-scope, but a test needs to be done at every access. As a result Jsigloo replaces all references to potentially intercepted variables by a call to a closure which is then inlined by Bigloo. This closure tests for the presence of a same-named member in the `with`-object, and executes the operation (either *get* or *set*) on the selected reference. Note that `with` constructs might be nested, and in this case the operation on the "selected reference" involves calling another function. This transformation (in a simplified version) is summarized in the following code snippet.

```
1: with(o) {
2:    x = y;
3: }
```

becomes

```
1: (let ((x-set! (lambda (val) (if (contains o x)
2:                                 (set! o.x val)
3:                                 (set! x val))))
4:       (y-get (lambda () (if (contains o y) o.y y))))
5:   (x-set! (y-get)))
```

This approach obviously introduces a performance penalty and together with the sometimes unexpected results (like the closure referencing different variables) a widely accepted recommendation is to avoid `with` completely [7].

4.4 Function Compilation

The function translation is the arguably most challenging part of a Javascript to Scheme compiler. Not only is Javascript a functional language where functions are frequently encountered, Scheme compilers usually optimize functions, and a good translation can reuse these optimizations. This section will restate the major differences between Javascript functions and Scheme procedures. We will then discuss each point separately, and detail how Jsigloo handles it. Bigloo is often unable to optimize Jsigloo's generic translation of functions, and the last part of this section presents Jsigloo's optimizations for functions.

Three primary features make the function translation from Javascript to Scheme difficult (for a more detailed discussion see Section 2):

- Every Javascript function can serve as method too. In this case every occurrence of the keyword `this` in the function's body is replaced by the object on which the function has been invoked. Otherwise `this` is replaced by the *global object*.
- It is possible to call every function with any numbers of arguments. Missing arguments are automatically filled with *undefined* and additional ones are stored in the *arguments* object.
- Javascript functions are objects.

Jsigloo's compilation of the `this` keyword is straightforward: When translating functions an additional parameter `this` is added

in front and all call-sites are adjusted: method calls pass the attached object as parameter, and function calls pass the *global object*.

Javascript functions can be called with any number of arguments and an early version of Jsigloo compiled functions to the intuitive form `(lambda (this . args) body)` to use Scheme's variable arity feature. Some measurements revealed that Bigloo was more efficient, if vectors were used instead of the implicit lists. At the call-sites a vector of all parameters is constructed, and then passed as second parameter after the `this`. A translated function is now of the following form: `(lambda (this args-vec) body)`.

Inside the function every declared parameter is then represented by a local variable of the same name. At the beginning of the procedure the local variables are either filled with their correspondent values from the arguments vector, or set to *undefined*. Figure 4 contains a simplified unhygienic version [8] of this process. The same figure shows the result for the declared parameters `a` and `b`.

```
1: `(let* ((len (vector-length vec))
2:         ,@(map (lambda (param-id count)
3:                  `(,param-id (if (> len ,count)
4:                                  (vector-ref vec ,count)
5:                                  'undefined)))
6:                param-list
7:                (iota (length param-list))))
8:    ,body)
```

```
1: (let* ((len (vector-length vec))
2:        (a (if (> len 0) (vector-ref vec 0) 'undefined))
3:        (b (if (> len 1) (vector-ref vec 1) 'undefined)))
4:   body)
```

Figure 4. the Jsigloo-extract at the top generates the code responsible for extracting the values out of the passed `vec`. The code at the bottom gets generated for the parameters `a` and `b`.

After the variable extraction Jsigloo creates the arguments object. As the `arguments`-entries are aliased with the parameter variables (`a` and `b` in the previous example) we use the same technique as for the `with` statement: the entries within the `arguments` object are actually closures modifying the local variables. Additional arguments access directly the values within the vector. Figure 5 demonstrates this transformation.

As has already been stated in Section 3, Javascript functions are mapped to the Bigloo class `Js-Fun`, which contains a field `fun` holding the actual procedure. Jsigloo's runtime library provides the procedure `make-js-function`, which takes a Scheme procedure with its arity and returns such an object. Jsigloo only needs to translate the bodies of Javascript functions, and generate code, that calls this runtime procedure with the compiled function as parameter . The returned object of type `Js-Fun` is compatible with translated Javascript objects. As the compiled function is now stored within an object, function calls are translated into a member retrieval, followed by the invocation of the received procedure.

The overhead introduced by these transformations is substantial: the compilation of the simple Javascript function `function f(a, b) {}` produces a Scheme expression of more than 20 lines, and the applied transformations are extremely counter-productive to Bigloo's optimizations. Storing the procedure in an object efficiently hides it from Bigloo's analyses. The Storage Use Analysis [15] (henceforth SUA), responsible for typing, and Bigloo's inlining pass are both powerless after this transformation. The arguments are then obfuscated by storing them in vectors, where Bigloo's constant propagation can not see them. When building the arguments objects they are furthermore accessed from inside a closure, which makes them slower to access.

```
1: `(let ((len (vector-length vec))
2:        (arguments (make-Arguments-object)))
3:    ,@(map (lambda (param-id count)
4:             `(if (> len ,count)
5:                  (add-entry arguments
6:                     (lambda () ,param-id)
7:                     (lambda (new-val)
8:                       (set! ,param-id new-val)))))
9:           param-list
10:          (iota (length param-list)))
11:   (let loop ((i ,(length param-list)))
12:      (if (> len i)
13:          (begin
14:            (add-entry arguments
15:               (lambda () (vector-ref vec i))
16:               (lambda (new-val)
17:                  (vector-set! vec i new-val)))
18:            (loop (+ i 1)))))
19:    ,body)
```

```
1: (let ((len (vector-length vec))
2:       (arguments (make-Arguments-object)))
3:   (if (> len 0)
4:     (add-entry arguments
5:       (lambda () a)
6:       (lambda (new-val) (set! a new-val))))
7:   (if (> len 1)
8:     (add-entry arguments
9:       (lambda () b)
10:      (lambda (new-val) (set! b new-val))))
11:  (let loop ((i 2))
12:    (if (> len i)
13:        (begin
14:          (add-entry arguments
15:            (lambda () (vector-ref vec i))
16:            (lambda (new-val)
17:               (vector-set! vec i new-val)))
18:          (loop (+ i 1)))))
19:  body)
```

Figure 5. the Jsigloo code at the top is responsible for the `arguments` creation in the emitted result. The bottom is generated for parameters `a` and `b`.

Jsigloo contains some optimizations addressing these issues. A simple one eliminates unnecessary lines: the creation of the arguments object is obviously only needed if the variable `arguments` is referenced inside the function. Otherwise Jsigloo just omits these lines.

In order to benefit from Bigloo's optimizations the indirect function calls need to be replaced by direct function calls wherever possible. Jsigloo's analysis is still relatively simple, but it catches the common case where declared (local or global) functions are directly called. The optimization is not yet correct though, and in its current form it needs to set an important restriction on the input: the given program must not modify any declared functions over the global object or in an `eval` statement. Section 6 discusses the necessary changes for the removal of this restriction.

Single Assignment Propagation (SAP) performs its optimization in two steps. First it finds all assignments to a variable and stores it in a set. Then it propagates constant values (including functions) of every variable that is assigned only once in the whole program.

Computing the definition-set is easy, but not trivial: Javascript automatically sets all local variables to *undefined* at the beginning of a function, and nearly every variable is hence modified at least twice. Once it is assigned to *undefined* and then to its initial value. Declared functions (global and local) are immediately set to their body and are hence treated accordingly. For all others a data-flow analysis needs to determine, if the variable might be used undefined. This analysis is mostly intraprocedural, and only needs one

pass. Some parts are however interprocedural as escaping variables cross function boundaries. Take for instance the following code:

```
1: function f() {
2:   var y = 1;
3:   var g = function() { return x + y + z; };
4:   var z = 2;
5:   g();
6:   var x = 3;
7:   return x + y + z;
8: }
```

Even though within `f` the variable `x` is read only after the definition in line *6*, the call at line *5* still uses the undefined variable. `y` on the other hand is always used after its first (and unique) definition. Usually these cases are difficult to catch, but SAP manages to find at least the most obvious ones: if a variable is defined before an anonymous function has been declared (as is the case for `y` in our example), the analysis does not add the implicit *undefined* definition to the variables definition set. SAP does hence correctly set `y`'s definition set to the assignment in line *2*, but will find two definitions for `z`. At the moment of `g`'s declaration `z` is still undefined, and as it is used within `g` the final definition set of `z` will hold the implicit *undefined*-definition and the assignment at line *4*.

The implicit *undefined* assignments are disturbing Bigloo's optimizations too. Whenever in doubt Jsigloo sets the variable to *undefined* at the beginning of a function. One of the first analyses Bigloo applies is the SUA-analysis, which detects the assignment of *undefined* and types the variable accordingly. Even if Bigloo is able to remove this assignment later on, it will not retype the variable, and misses precious optimization opportunities.

Once the definition-set has been determined, a second pass propagates "single assignments". If a variable has only one assignment in its definition set, and this assignment sets the variable to a constant value or a Javascript function, all occurrences of this variable are replaced by either the constant, or by a reference to this function. In our example the line *7* still uses `x` and `z`, as their definition-sets contain more than one assignment. The optimization transforms our previous example into the following code:

```
1: function f() {
2:   var y = 1;
3:   var g = function() { return x + 1 + z; };
4:   var z = 2;
5:   anonymous_g();
6:   var x = 3;
7:   return x + 1 + z;
8: }
```

Wherever the backend finds direct function-references it is now able to optimize the call. Instead of extracting the procedure from the function object it can use the function-reference. The previous creation of function objects must first be modified to allow access to the procedure:

```
1: (set! direct_f (lambda (this vec) body))
2: (set! f (make-js-function direct_f 2))
```

In our benchmarks and test-cases 27% of all function calls could be replaced by direct function calls after this analysis.

Wherever Jsigloo is able to replace the indirect calls with direct calls it can also improve the parameter passing. The function's signature provides the expected number of arguments, and the parameters do not need to be hidden in a vector anymore. If there are missing arguments, they can already be filled with *undefined* constants at compile-time. An additional `arg-nb` parameter passes the original number of arguments, which is needed for the creation of the arguments object. The last argument finally contains additional arguments, that have not been mapped to direct parameters. They will be used during `arguments` creation, too. Obviously the generic call needs to be adapted too, and the parameter-extraction of figure 4 is lifted into the procedure passed to the `make-js-function`.

Many functions do not use `this` and in this case the first argument can be removed. The same is of course true for `arg-nb` and `rest-vec`, which are only needed, if the function uses the arguments-object. Our running example is finally transformed into the following code:

```
1: (set! direct_f (lambda (a b) body))
2: (set! f (make-js-function
3:             (lambda (this vec)
4:               (let* ((len (vector-length vec))
5:                      (a (if (> len 0)
6:                             (vector-ref vec 0)
7:                             'undefined))
8:                      (b (if (> len 1)
9:                             (vector-ref vec 1)
10:                            'undefined)))
11:                (direct_f a b))
12:            2))
```

Applying these optimizations to the well known Fibonacci function let the size of procedure drop from more than 75 to about 20 lines[9], and reduce execution time by a factor of more than 20.

5. Performance

	Ack	Fib	Meth	Nest	Tak	Hanoi
Jsigloo J	1931	443	185	**898**	28	424
Rhino	1042	666	155	973	55	619
Jsigloo C	**513**	**368**	84	1060	**11**	**368**
Konqueror	-	17183	262	15478	593	21049
Firefox	-	3179	227	1808	79	2762
NJS	-	767	**23**	1481	25	734

Jsigloo is not yet finished, and the given benchmarks (see the appendix for the sources) are therefore just indications. As we wanted to be able to run our benchmarks on most existing Javascript implementations we decided to move the time-measurement into the benchmark itself. This way it was possible to benchmark Internet browsers too. At the same time we lost the start-up overhead, and the more precise measurement of the Linux kernel. All times have been taken under a Linux 2.6.12-nitro on an AMD Athlon XP 2000+, and are expressed in Milliseconds. We used Sun's JDK 1.4.2.09 (HotSpot Client VM, mixed mode) and GCC 3.4.4. We ran every benchmark at least three times, and report the fastest measured time here. Konqueror [2], Firefox [1] and NJS [3] where not able to complete Ack (stack overflows) and do not have a time for this benchmark.

"Jsigloo J" uses Bigloo's JVM backend, whereas "Jsigloo C" targets C, followed by a compilation to native code. "Rhino", in version 1.6R2RC2, compiles Javascript directly to JVM bytecode and competes hence with "Jsigloo JVM". The fastest time on the JVM machine is underlined. "Konqueror" 3.4.2, "Firefox" 1.0.6 and "NJS" 0.2.5 are all interpreters (even though NJS was allowed to precompile the Javascript code into its bytecode format) and are compared to "Jsigloo C". The fastest time is in bold.

During the development these benchmarks have been (and are still) used to pinpoint weak spots of Jsigloo, which were then improved. One of the first benchmarks has been Fibonacci, which explains Jsigloo's good results in some of the other call-intensive benchmarks (Hanoi and Tak). "Nest", as the name hints, increments a number within nested loops. We verified our results and for this benchmark the Java version is actually faster than the native

[9] We are well aware, that this is still far away from a standard 5 lines implementation, but most of the resting lines are redundant `lets`, `begins` or `#unspecified` which are easily removed by Bigloo.

C version. The JVM version of Ackermann is still slower than Rhino's code, but we have pinpointed the source of inefficiency and a generic transformation brings the time for "Ack" down to the same level as Rhino. "Meth" on the other hand makes heavy use of anonymous functions and objects, and this part of Jsigloo is not yet optimized at all.

Note, that Jsigloo is not conformant to the ECMA specification (see Section 6), and has therefore an unfair advantage over the competitors. Some tests showed that Fibonacci's execution time would double if the global object was treated correctly. Other experiences however confirmed, that for instance a fully optimizing Rhino is not conformant either, and especially the *global object* is equally ignored.

6. Future Work

Jsigloo is not finished. Several Javascript features have not yet been implemented and some parts of Jsigloo are not conformant to the ECMA specification. From the more than 10 runtime objects, only two have been written until now (in particular the `Boolean`, `String`, `Number` and `Date` objects are still missing). Due to limitations in the used lexer-generator, some syntactic sugar is missing too (Javascript's automatic semicolon insertion and its regular expression literals).

At the moment Jsigloo does not handle the global object correctly either. It is not possible to modify global variables over an object, and function calls receive a standard `Object` as `this`. We intend to fix this shortcoming by adding two strategies:

- a "correct" solution using a special global object, that holds closures. Whenever a field is modified the closure automatically updates the real global object. (Inversely reading from the global object automatically redirects to the real global variable). A similar strategy is already being used for the `arguments` object and the `with` translation.
- a fast implementation which disallows the use of the global object. Every access to the global object throws an exception.

The `eval` function is missing too. Javascript's and Scheme's `eval` specification are different and incompatible, but Bigloo provides some extensions to Scheme's `eval` which should allow the implementation without too much trickery.

Once either the `eval`-function or the global object is correctly implemented, the SAP optimization of Section 4.4 needs to be adapted. Functions that are visible to `eval` statements and global functions might not be called directly anymore.

Finally the number representation needs to improved. Javascript numbers are mapped to Scheme doubles. In R^{5}RS [12] doubles do not provide enough functionality to correctly represent Javascript numbers, but R^{6}RS will extend Scheme's number specification, and we will revisit this topic once R^{6}RS has been released.

7. Conclusion

We presented in this paper Jsigloo, a Javascript to Scheme compiler we implemented during the last five months. Together with Bigloo it compiles Javascript to Java byte-code, C, or .NET CLI. In the introduction we listed the features Jsigloo should have. We wanted the compiler to be small. Jsigloo is not very big, but with about 30.000 lines of Scheme code Jsigloo is not small anymore. It is still easy to maintain the project, but the effort required is higher than we hoped it would be. Jsigloo's size is explained by the optimizations we integrated in Jsigloo, and preliminary benchmarks show that Jsigloo/Bigloo has the potential to be as fast as the fastest existing Javascript compilers. As Jsigloo uses Bigloo it interfaces with all languages Bigloo interfaces and excels in this area. Furthermore, if Bigloo improves, Jsigloo/Bigloo will improve too.

Despite Javascript's resemblance to Scheme, we could not take full advantage of all Bigloo optimizations and needed to implement additional optimization passes. SAP (Section 4.4) enhances direct method calls, a `bind-exit`-removal pass (Section 4.1) eliminates unnecessary (but currently expensive) `bind-exits`, and typing (Section 3) improves the ubiquitous conversions of Javascript and helps several Bigloo optimization by providing Javascript-specific type information.

Compiling to Scheme and using an efficient existing Scheme compiler did not fulfill all our expectations, but still yielded an interesting compiler. Once all missing features are implemented Jsigloo may be an attractive alternative to all other Javascript compilers.

[1] **http://www.mozilla.org/products/firefox/**.
[2] **http://www.konqueror.org/**.
[3] **http://www.njs-javascript.org/**.
[4] **http://www.caucho.com/articles/990129.xtp**.
[5] **http://www.mozilla.org/rhino/**.
[6] Bobrow, D. and DeMichiel, L. and Gabriel, R. and Keene, S. and Kiczales, G. and Moon, D. – **Common lisp object system specification** – special issue, Sigplan Notices, (23), Sep, 1988.
[7] D. Flanagan – **JavaScript - The definitive guide** – *O'Reilly & Associates*, 2002.
[8] Dybvig, K. and Hieb, R. and Bruggeman, C. – **Syntactic abstraction in Scheme** – Lisp and Symbolic Computation, 5(4), 1993, pp. 295–326.
[9] ECMA – **ECMA-262: ECMAScript Language Specification** – 1999.
[10] Flatt, M. and Krishnamurthi, S. and Felleisen, M. – **Classes and Mixins** – Symposium on Principles of Programming Languages, Jan, 1998, pp. 171–183.
[11] Goldberg, A. and Robson, D. – **Smalltalk-80: The Language and Its Implementation** – *Addison-Wesley*, 1983.
[12] 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.
[13] Martin Abadi and Luca Cardelli – **A theory of objects** – *Springer*, 1998.
[14] Muchnick, S. – **Advanced Compiler Design Implementation** – *Morgan Kaufmann*, 1997.
[15] Serrano, M. and Feeley, M. – **Storage Use Analysis and its Applications** – 1fst Int'l Conf. on Functional Programming, Philadelphia, Penn, USA, May, 1996, pp. 50–61.
[16] Serrano, M. and Weis, P. – **1+1=1: an optimizing Caml compiler** – ACM Sigplan Workshop on ML and its Applications, Orlando (Florida, USA), Jun, 1994, pp. 101–111.

8. Appendix

Benchmarks

Ackermann

```
1: function ack(M, N) {
2:     if (M == 0) return(N + 1);
3:     if (N == 0) return(ack(M - 1, 1));
4:     return(ack(M - 1, ack(M, (N - 1))));
5: }
6:
7: ack(3, 8);
```

Fibonacci

```
1: function fib(i) {
2:    if (i < 2)
3:       return 1;
4:    else
5:       return fib(i-2) + fib(i-1);
6: }
7: fib(30);
```

Method Calls

```
function methcall(n) {
   function ToggleValue () {
      return this.bool;
   }
   function ToggleActivate () {
      this.bool = !this.bool;
      return this;
   }

   function Toggle(start_state) {
      this.bool = start_state;

      this.value = ToggleValue;
      this.activate = ToggleActivate;

   }

   function NthToggleActivate () {
      if (++this.count > this.count_max) {
         this.bool = !this.bool;
         this.count = 1;
      }
      return this;
   }

   function NthToggle (start_state, max_counter) {
      this.base = Toggle;
      this.base(start_state);
      this.count_max = max_counter;
      this.count = 1;

      this.activate = NthToggleActivate;

   }
   NthToggle.prototype = new Toggle;

   var val = true;
   var toggle = new Toggle(val);
   for (i=0; i<n; i++) {
      val = toggle.activate().value();
   }
   var tmp = (toggle.value() ? "true"
                             : "false");

   val = true;
   var ntoggle = new NthToggle(val, 3);
   for (i=0; i<n; i++) {
      val = ntoggle.activate().value();
   }
   return (tmp + " " +
             (ntoggle.value() ? "true"
                              : "false"));
}

methcall(10000);
```

Nested Loops

```
function nested(n) {
   var x=0;
   var a=n;
   while(a--) {
      var b=n; while(b--) {
         var c=n; while(c--) {
            var d=n; while(d--) {
               var e=n; while(e--) {
                  var f=n; while(f--) {
                     x++;
                  }
               }
            }
         }
      }
   }
   return x;
}

nested(14);
```

Tak

```
function tak(x, y, z) {
   if (!(y < x))
      return(z);
   else {
      return (
         tak (
            tak (x-1, y, z),
            tak (y-1, z, x),
            tak (z-1, x, y)
            ));
   }
}

tak(18, 12, 6);
```

Towers of Hanoi

```
function towers(nb_discs, source, dest, temp) {
   if (nb_discs > 0) {
      return towers(nb_discs - 1,
                    source,
                    temp,
                    dest)
             + 1
             + towers(nb_discs - 1,
                      temp,
                      dest,
                      source);
   }
   return 0;
}

towers(20, 0, 1, 2);
```