Vrije Universiteit Brussel

Implementing a Performant Scheme Interpreter for the Web in asm.js

Van Es, Noah; Stiévenart, Quentin; Nicolay, Jens; D'Hondt, Theo; De Roover, Coen

Published in:
Computer Languages, Systems and Structures

DOI:
10.1016/j.cl.2017.02.002

Publication date:
2017

Link to publication

Citation for published version (APA):
Van Es, N., Stiévenart, Q., Nicolay, J., D'Hondt, T., & De Roover, C. (2017). Implementing a Performant Scheme Interpreter for the Web in asm.js. *Computer Languages, Systems and Structures*, *49*, 62-81. https://doi.org/10.1016/j.cl.2017.02.002

Copyright
No part of this publication may be reproduced or transmitted in any form, without the prior written permission of the author(s) or other rights holders to whom publication rights have been transferred, unless permitted by a license attached to the publication (a Creative Commons license or other), or unless exceptions to copyright law apply.

Take down policy
If you believe that this document infringes your copyright or other rights, please contact openaccess@vub.be, with details of the nature of the infringement. We will investigate the claim and if justified, we will take the appropriate steps.

Download date: 20. Apr. 2024

Implementing a Performant Scheme Interpreter for the Web in asm.js

Noah Van Es[a], Quentin Stievenart[a], Jens Nicolay[a], Theo D'Hondt[a], Coen De Roover[a]

[a] *Software Languages Lab, Vrije Universiteit Brussel, Belgium,*
{noahves,qstieven,jnicolay,tjdhondt,cderoove}@vub.ac.be

Abstract

This paper presents the implementation of an efficient interpreter for a Scheme-like language using manually written asm.js code. The asm.js specification defines an optimizable subset of JavaScript which has already served well as a compilation target for web applications where performance is critical. However, its usage as a human-writable language that can be integrated into existing projects to improve performance has remained largely unexplored. We therefore apply this strategy to optimize the implementation of an interpreter. We also discuss the feasibility of this approach, as writing asm.js by hand is generally not its recommended use-case. We therefore present a macro system to solve the challenges we encounter. The resulting interpreter is compared to the original C implementation and its compiled equivalent in asm.js. This way, we evaluate whether manual integration with asm.js provides the necessary performance to bring larger applications and runtimes to the web. We also refactor our implementation to assess how more JavaScript code can cohabit with asm.js code, improving maintainability of the implementation while preserving near-native performance. In the case of our interpreter, this improved maintainability enables adding more complex optimizations. We investigate the addition of function inlining, for which we validate the performance gain.

Keywords: Optimization, JavaScript, asm.js, Interpreters, Performance
2000 MSC: 68N15, 68N20

---

1. Introduction

Our study starts with the implementation of an efficient interpreter for a Scheme-like language in JavaScript. Using JavaScript as a host language to build an interpreter enables a new language on the web that inherently becomes available to millions of users, as nearly each platform today is equipped with a browser that includes a JavaScript virtual machine. In terms of performance, however, a high-level language such as JavaScript does not meet the necessary requirements for an efficient language implementation. JavaScript in particular has a highly dynamic nature that does not allow for efficient static analysis and optimization. Typically, a JavaScript engine features a JIT compiler complimented with other dynamic optimizations [11] to achieve acceptable performance in a dynamic setting. However, for an efficient language implementation, we do not want to compromise performance in the host language. We therefore turn to a more optimizable, restricted subset of JavaScript, asm.js [12], as a means to improve the efficiency of performance-critical JavaScript applications such as an interpreter. By eschewing many of JavaScript's dynamic features, asm.js promises to deliver near-native performance on the web. asm.js limits the JavaScript to numerical types, top-level functions, and one large binary heap. With the addition of static typing, an optimizing JavaScript engine is able to compile and optimize asm.js code ahead of time. Because it remains a strict subset of JavaScript, existing JavaScript engines are automatically backward compatible with asm.js.

At this time, asm.js is mainly used as a compilation target. Developers start with an existing C/C++ application which they can then efficiently port to the web by compiling it to asm.js using Emscripten [21]. Our approach, however, is different. We start with an existing JavaScript implementation and attempt to improve its performance by integrating asm.js. The idea here is that using asm.js for the core components of a JavaScript application improves the overall performance of that application. Such a mix of JavaScript and asm.js is possible, since the latter can interface with external JavaScript code. We therefore apply this strategy to our interpreter and rewrite its most crucial components (such as the memory management) into asm.js. As a result, we iteratively refactor our application by lowering down its modules into asm.js one-by-one. This produces a series of successive implementations, where we expect to see an improvement in performance for each iteration. We then benchmark each such milestone to measure the actual performance impact of this asm.js integration process.

Another point of interest is that we write this asm.js code by hand. This is unconventional, as asm.js mainly serves as a compilation target and is therefore not designed to be written manually. As a result, we encounter several challenges in our attempt to do so. For instance, we notice a severe lack of readability and maintainability in asm.js applications. These are not really issues for a compiler, but they do complicate the usage of handwritten asm.js at larger scales. Furthermore, asm.js can be considered a low-level language, offering similar functionality as C in a JavaScript syntax. All data also has to be encoded into numbers and bytes, as asm.js only supports numerical types. The top-level array holding these numbers has to be managed manually, since asm.js does not support any form of garbage collection.

These challenges, however, do not limit the possibilities of asm.js. To deal with the restrictions in readability and maintainability, we propose a solution using macros. By using a specialized macro expander, many practical limitations can be hidden into a more convenient syntax. Such a preprocessor enables writing certain parts of the asm.js code indirectly as a high-level, domain-specific language, and therefore defines a more human-writable dialect of the language. We illustrate this topic further in Section 3.1.

Afterwards, we take a step back and compare our handwritten implementation to the conventional strategy of compiling an existing C application into asm.js. We also compare the performance of our implementation with that of an equivalent version as well as the native implementation itself. In order to make this comparison, we first add some interpreter optimizations directly into the asm.js code. This also enables us to evaluate the maintainability of macro-enabled asm.js applications. The impact on development effort can then determine whether it is worth to write such asm.js code by hand.

Having JavaScript and asm.js code cohabit introduces new challenges for the interoperability between the two. We identify the limitations and propose a solution that introduces a level of indirection through *handles*. This allows us to refactor the compiler component of our interpreter back in plain JavaScript, for improved maintainability without sacrificing performance. This enables adding more complex optimization to the compiler with lower effort, and we investigate this by adding function inlining.

In conclusion, we advocate using asm.js for performance-critical components of an application, while using more high-level JavaScript for the components that are not performance-critical, instead of writing most of the application in asm.js.

We focus our study on the implementation of an interpreter for two rea-

sons. First, we argue that writing an interpreter is a flexible approach to support a new language on a platform, in this case the web browser. In general, writing an interpreter requires less effort than implementing a compiler for the same language. Second, although easier to obtain, interpreters generally are not as fast as compiled code, and their relative performance becomes worse when the host language itself is (partly) interpreted. It is therefore interesting to examine whether and how it is possible to improve their performance by moving to a more low-level host language, asm.js. These reasons make the implementation of an interpreter for a dynamic language an excellent target for our experience report. Our work therefore presents a systematic approach that starts with a simple interpreter and later integrates asm.js to achieve an efficient web implementation of a dynamic language.

Overall, this paper highlights the development efforts and resulting performance improvements of integrating asm.js into an existing application. It provides an experience report of our particular usage of asm.js and makes the following contributions:

- An overview of the performance impact that can be achieved by integrating asm.js into existing projects.
- A solution by introducing a macro preprocessor to improve readability, maintainability and performance when writing asm.js code by hand.
- A novel approach to achieve an efficient web implementation of a dynamic language by integrating asm.js into a simple interpreter. We illustrate this using our own handwritten implementation of a garbage-collected Scheme interpreter, in which we integrated asm.js to enable good performance on the web.
- A comparison between two different strategies using either handwritten or compiled asm.js to port runtimes and codebases to JavaScript.
- A solution to have JavaScript and asm.js code cohabit at runtime without leading to corrupt references.

This is an extended version of previous work [20]. This journal paper extends the conference publication in multiple ways.

- In the original paper, most of the components are lowered down into asm.js in order to achieve near-native performance for a Scheme inter-

preter written in JavaScript. Notably, the compiler is written in low-level asm.js, and is not easy to extend. In this version, we investigate how to bring maintainability of the compiler back, by rewriting some components of our final version of the interpreter into plain JavaScript. We identify that sharing pointers between asm.js and JavaScript is problematic, and introduce handles to solve this problem. We end up with an interpreter where the performance-critical components are still written in low-level asm.js, while other components such as the compiler are implemented in high-level JavaScript. We demonstrate that we still achieve the same near-native performance in this version.

- The usage of high-level JavaScript in the components that are not performance-critical allows us to extend our compiler with more optimizations. We describe how we added function inlining to the interpreter, and discuss the performance impact of this feature.
- We discuss related work on WebAssembly, a future open standard inspired by asm.js destined to be a portable and efficient code format for the web.

2. Setting

We apply the strategy of integrating asm.js to the field of interpreters, where performance is usually a critical requirement. The language that the interpreter executes is Slip[1], a variant of Scheme. An implementation of the language is available in C and is being used in a course on programming language engineering[2]. It served as the basis for the design and implementation of our own interpreter, named slip.js[3].

The semantics of Slip [5] closely resembles that of Scheme. Differences are subtle and mostly limited to the usage of certain natives and special forms. Slip intends to go back to the original roots of the language and throws away many of the recent, non-idiomatic additions that are targeted more towards industrial engineering rather than an academically designed language. For instance, it considers `define` to be the most appropriate construct for variable binding, and only provides a single `let`-form. Slip also

[1]Simple Language Implementation Platform (also an anagram for LISP).
[2]`http://soft.vub.ac.be/~tjdhondt/PLE`
[3]`https://github.com/noahvanes/slip.js`

enforces left-to-right evaluation of arguments, since not doing so is usually related to an implementation issue rather than a sound design choice.

The first version of the interpreter uses plain JavaScript only. It is ported over from a metacircular implementation of Slip and serves as a useful prototype that can be gradually lowered down to asm.js. Doing so enables the design of an efficient interpreter in a high-level language, without dealing with the complexity of asm.js as yet.

2.1. Stackless design

The initial interpreter design employs continuation-passing style (CPS) to build up a custom stack in the heap instead of using the underlying JavaScript stack. We call the result a "stackless" design because it does not rely upon the runtime stack of the host language to store the control context. As a consequence, stack overflows are no longer caused by exceeding the JavaScript stack size, but depend on the available heap memory instead.

The custom stack can be seen as a concrete representation of the current continuation. Having full control over evaluation contexts facilitates the implementation of advanced control constructs such as first-class continuations [18, Ch. 3] in our language. It also makes it easier to iterate over the stack for garbage collection, since the stack can indirectly summarize all 'active' references to allocated objects on the heap.

As a result of using CPS, all functions calls appear in tail position. With the arrival of proper tail recursion in the current ES6 specification of JavaScript and in WebAssembly (cf. Section 9), these tail calls should be properly optimized so that they do not cause any stack overflows. However, the feature is not yet available in all mainstream browsers and other runtimes, even when they support most of the other features of the ES6 specification. As a subset of JavaScript, asm.js currently does not offer this feature either. Therefore we use a trampoline to avoid uncontrolled stack growth. This ensures that our tail calls do not grow the JavaScript stack.

More formally, a trampoline can be defined as a function that keeps on calling its argument thunk until it becomes *false* [9, p. 158]. Such a trampoline loop can be implemented in asm.js (or JavaScript) using an iterative construct as illustrated below.

```
function run(instr) {
  instr=instr|0;
  for(;instr;instr=FUNTAB[instr&255]()|0);
}
```

Using this iterative loop, each function returns the function table index of the next function to be called. As discussed, we build up our own stack in the heap to store the continuation frames.

Note that, as a low-level language, asm.js is statically typed. It uses the bitwise OR operator as a type annotation for 32-bit integers. For example, the line `instr=instr|0;` in the previous code snippet indicates that the type of parameter `instr` is a 32-bit integer. Using the bitwise OR operator instead of custom type annotation allows asm.js to remain fully backward compatible with the existing JavaScript specification. JavaScript engines that do not specifically optimize asm.js (AOT) can execute this statement without changing the semantics of a program. Similarly, the lack of first-class functions in asm.js forces the usage of an explicit function table `FUNTAB` to implement the trampoline. The use of this function table and the bitwise AND operator in the index is further explained in Section 3.2.

2.2. Optimized memory model

Instead of relying on the underlying memory management of JavaScript, the interpreter allocates its own memory chunks. It is accompanied by an iterative mark-and-sweep garbage collector. The memory model takes over many optimizations from the original C implementation, such as the usage of tagged pointers. This allows us to inline simple values and avoids indirections to unnecessary memory chunks.

Using a custom chunked memory model is a necessary provision, since asm.js does not provide any support for objects or garbage collection. Moreover, Slip keeps all its runtime entities in a single large address space. This makes it easier to map this heap onto the single memory array that is used to store values in asm.js.

2.3. Register-machine architecture

Experiments we conducted with the C implementation of Slip showed that using explicit registers can result in significant performance improvements. We therefore opted for a register-machine architecture in asm.js as well. According to the specification [12], asm.js is able to store the contents of these registers efficiently by mapping them to raw 32-bit words. A register-machine architecture is possible in our implementation because the evaluation process is in CPS and therefore only uses tail calls. Such an iterative process is known to require only a fixed amount of iteration variables, and in our implementation around twenty dedicated registers are available for

this purpose. Each register serves a specific purpose and is shared throughout the entire interpreter infrastructure. For instance, the `KON`-register stores the current continuation, whereas the `FRM` and `ENV` hold the current frame and environment.

With only tail calls and no local variables or arguments, the host stack of JavaScript remains untouched. This facilitates the implementation of garbage collection, since all local variables and arguments currently in scope can be accessed through the registers and heap-allocated stack. All references in the registers are automatically updated after triggering a garbage collect.

2.4. Imperative style

Finally, due to the transformation to CPS and the usage of registers, the interpreter follows a very low-level, imperative style. In fact, the evaluator shows a lot of similarity with the explicit-control evaluator from the SICP handbook [1, pp. 547–566]. Having such code in the initial prototype makes the transition to the low-level constructs of asm.js easier later on.

3. asm.js integration

3.1. Integration process

The integration process lowers down the modules in the interpreter to asm.js, starting with the most performance-critical ones. Each iteration is expected to improve performance of the previous one by refactoring another component. Completing this process then results in an interpreter that is almost entirely written in asm.js.

System decomposition. Figure 1 shows the interpreter pipeline. The program input is first preprocessed by two analyzers, a *parser* and a *compiler*. The former constructs a basic abstract syntax tree (AST), while the latter performs some optimizations at compile-time. The compiler employs a rich abstract grammar that is able to provide more static information to the interpreter than the original AST. This takes away some of the processing work for the evaluator and thus improves runtime performance. A pool is used to store all the symbols for enabling efficient comparisons using pointer equivalence. The resulting AST is then interpreted by the evaluator, which forms the core of the interpreter. Finally, a printer presents resulting values appropriately. Two other important modules are the abstract grammar and the memory

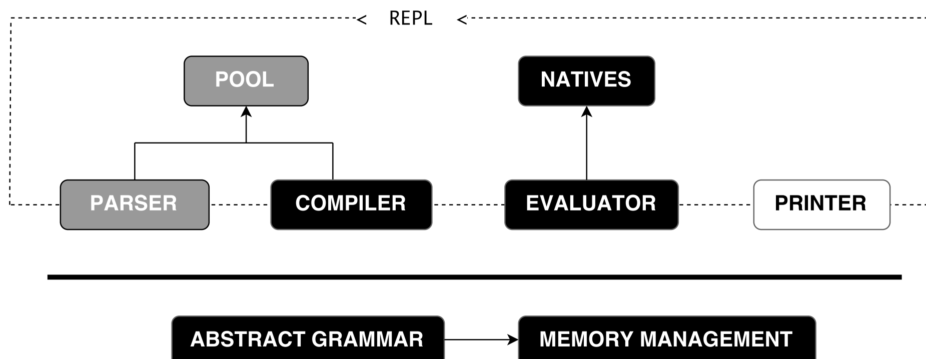

Figure 1: Interpreter pipeline.

management. The abstract grammar module defines the AST nodes that are produced by the compiler and the parser. Slip.js uses these AST nodes extensively as a unified abstract grammar for both values and expressions. The memory management is responsible for managing these AST nodes in the global heap. It provides procedures to allocate memory regions (which we refer to as *chunks*) for AST nodes in the heap. The memory management also provides procedures for garbage collection to automatically free unused chunks when allocation requires more memory. As indicated by the heavy line in Figure 1, these critical modules form the foundation for the entire interpreter infrastructure.

Milestones. While lowering down the modules into asm.js, four different milestones were identified. `asm0` refers to the original prototype version in plain JavaScript. It makes no use of asm.js. `asm1` is the first step in the integration process. It lowers down the memory management into asm.js, as it is one of the most critical components in the interpreter. `asm2` also refactors the abstract grammar and merges it with the memory management into a single asm.js module. `asm3` is the biggest leap in the refactoring process. Due to limitations in working with multiple asm.js and non-asm.js modules (cf. Section 5) most of the other components are lowered down into a single asm.js module at once. Figure 2 shows how much of the total code in each version is written in asm.js. These ratios are not representative of the actual contribution of asm.js, as some components are more important than others. Instead, they merely visualize how the asm.js and JavaScript mix evolved over time.

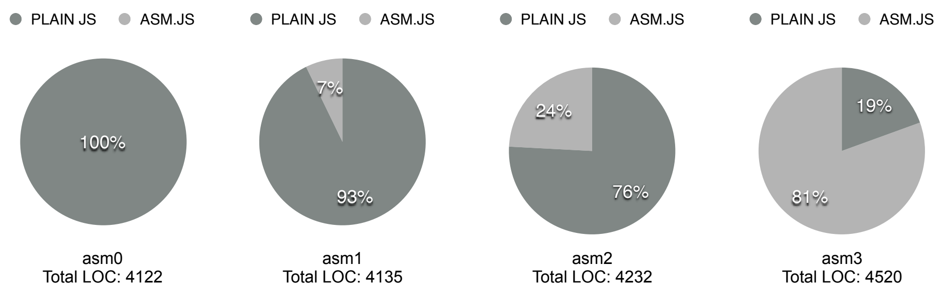

Figure 2: asm.js integration process.

Overview. The integration process starts with the most performance-critical components of our application, in this case the memory management and abstract grammar. Afterwards, other core modules such as the evaluator and natives are lowered down into asm.js as well. The colors in Figure 1 indicate the final result after all the refactoring up to `asm3`. Modules colored black are completely written in asm.js, whereas the grey boxes indicate the presence of regular JavaScript. The only component that does not use any asm.js is the printer, hence the white box in the diagram.

The memory management and the abstract grammar are the first components lowered down into asm.js. This choice is motivated by the fact that they are frequently used throughout the interpreter, and therefore have a considerable impact on performance. This transition is also quite gentle, since the memory model was already designed at the bit-level in the prototype. Similar performance considerations also hold for the evaluator and the natives that make up the core of the interpreter and should therefore be optimized as much as possible. Having the compiler and parser partially lowered down into asm.js has more to do with convenience to facilitate interaction with other modules, rather than true performance concerns. They are only invoked once before execution and are not considered a bottleneck in the interpreter. For this reason, the parser is not entirely written in asm.js. String manipulation is also much easier in traditional JavaScript, so the parser relies on a separate external module to iterate over the input program string. The same argument also holds for the pool. Designing an efficient map between Slip symbols (i.e. strings) and their respective index in the pool is not trivial in asm.js. This is much easier in JavaScript, since we can simply use an object for this map. As a consequence, the pool module still communicates with external JavaScript to query the map for existing symbols. The printer uses no asm.js at all. It is invoked only once after evaluation to print the

resulting output string.

3.2. Macros to the rescue

Manually integrating asm.js into the interpreter turned out to have some practical limitations. Its low-level style enforces duplication and lacks proper mechanisms for abstraction. This results in poor maintainability for non-trivial applications, such as our interpreter. For this reason, we started to rely on the generated nature of asm.js and turned to macros. Macros help to avoid low-level asm.js constructs by providing a language that is more readable and writable by humans.

We use *sweet.js*[4] as a macro expander. It is an advanced, hygienic macro expander for JavaScript (and therefore also asm.js) that provides a macro framework similar to how Scheme provides `syntax-case` and `syntax-rule` macro definitions. It can be run against a source file with macro definitions to produce a pure JavaScript file with all macros expanded. We discuss some of our use-cases of macros.

Constants. Constants are useful in the interpreter for many different purposes, such as the efficient enumeration of tags for the different abstract grammar items. However, asm.js only offers mutable variables, which are comparably less efficient and cannot be used as compile-time constants (e.g. in the branches of a switch statement). We therefore define a macro `define` for introducing constants, which works similarly to the `#define` preprocessor directive in C. The multi-step expansion of macros in sweet.js can be used to implement `define` as a macro that in turn defines a new macro for a constant's name.

```
macro define {
     rule { $nam $val } => {
          macro $nam {
            rule {} => {$val}
          }
     }
}
```

AST nodes. Another macro, called `struct`, is more domain-specific and enables us to concisely define the abstract grammar of the interpreter, similar

[4]`https://www.sweetjs.org`

to how one defines a `struct` type in C. The macro transforms the description of an AST node into an actual implementation used by slip.js. It automatically generates a constructor and accessors that are implemented as macros themselves for efficiency (cf. Section 4.2). The generated code deals with all necessary asm.js annotations and calculates the required amount of bytes to be allocated through the memory manager. Its main purpose therefore is to improve readability and flexibility, and avoid duplication.

Function table. With the transformation to CPS (cf. Section 2.1) and the register-machine architecture (cf. Section 2.3), it becomes natural to think of functions and calls as instruction sequences and jumps between them. We therefore introduce macro `instructions` that enables expressing control in the interpreter using labels and gotos. While this decision may appear questionable at first, it is no different from having zero-argument functions that are only called in tail position. The main reason to use a macro for expressing control is because functions are not first-class in asm.js, making it impossible to return them to the trampoline or store their reference somewhere in the heap. A numerical encoding of functions therefore is required to simulate function pointers. For this purpose, we employ a function table, which asm.js allows as long as all functions have an identical signature. The index of a function stored in the function table can then be used as a pointer to that function. However, managing all these pointers manually becomes unmaintainable at larger scales, as it becomes hard to associate each function with its corresponding index. Moreover, asm.js requires that the size of the function table is a perfect power of two, so that it can be indexed using a bitwise AND operator with a full bit mask instead of doing additional boundary checks at run-time. The `instructions` macro therefore takes the following steps:

1. it transforms all labels into zero-argument functions with the corresponding instruction sequence as body,

2. it puts all these functions into a function table and uses padding to increase its size to a power of two, and

3. it defines a constant to replace each function name with a function pointer according to the index that function got in the function table.

This last step is done using the `define` macro, so that each function name

automatically gets replaced with its index into the function table. Padding involves adding extra `nop` functions at the end of the function table.

4. Optimization

The previous sections discussed how the interpreter was first designed in a high-level language (JavaScript), and then systematically translated into a low-level subset (asm.js). In order to evaluate the maintainability of hand-written, macro-enabled asm.js applications, however, it is also interesting to add new functionality directly into that asm.js code. We therefore apply a series of optimizations to `asm3` and produce an improved version called `asm4`. We then put this final version into perspective by comparing its performance with the original C implementation in Section 5.2.

4.1. Interpreter optimizations

Most of the optimizations included in `asm4` are traditional interpreter optimizations [18, Ch. 6]. We highlight some of them below.

Lexical addressing. The first major improvement is the implementation of lexical addressing. Slip, as a dialect of Scheme, employs static binding, where free variables are looked up in lexical environments. The exact frame and offset where a variable can be found therefore can be determined without executing the program. Lexical addressing builds up a static environment at compile-time, and replaces each variable occurrence with the index of the frame in the environment and the variable's offset into that frame (also known as lexical addresses or de Bruijn indices [4]). This process is done through a new component, the dictionary, called from the compiler. The evaluator can then use these indices to access a variable in constant time instead of looking up the variable at runtime.

Rich abstract grammar. Other large performance improvements are achieved by further diversifying the abstract grammar and detecting more static features at compile-time. Doing so provides more information to the evaluator and further improves runtime performance of the interpreter. For instance, any proper Slip implementation should support tail call optimization. Whether a function call is in tail-position is a static feature, and therefore it makes sense to detect such tail calls at compile-time. Another major heuristic observation is that most function applications have a simple expression (such as a variable) in operator position. Simple expressions can simplify

the operation of the interpreter, because they can be immediately evaluated. Unlike compound expressions, an interpreter does not need to step into simple expressions, and "remember" where to continue after evaluating a subexpression by pushing a continuation on the stack. Therefore, detecting and marking applications with simple operators with a special AST node at compile-time enables optimization of their execution at run-time.

Tagged pointers. In order to avoid unnecessary chunks and indirections, the initial prototype already employs tagged 32-bit pointers to inline small values. These include small integers, lexical addresses of variables and special values like booleans and the empty list (null). More precisely, if the LSB is set, the other 31 bits can hold any immediate value instead of an actual pointer. Further elaborating this strategy using a Huffman encoding of tags makes the usage of these bits more efficient. This enables more values to be inlined, which reduces memory access even further, while still maintaining a reasonable value range for each immediate type. For instance, small integers only use two bits for their tag, leaving the remaining 30 bits free to represent a numerical value. Local variables on the other hand require five bits to recognize their tag. This still gives them a substantial range of 2^{27} values to indicate the offset of the variable in the frame.

Enhanced stackless design. The stackless design from Section 2.1 uses a continuation-passing style in conjunction with a trampoline to avoid growing the underlying JavaScript stack. Using our own stack simplifies the implementation of garbage collection and advanced control constructs such as first-class continuations. However, returning to the trampoline causes a small performance overhead, as each jump requires to return an index to the trampoline and lookup the function in the function table. We therefore allow some jumps to call the corresponding function directly, instead of returning to the trampoline first. Such a call will make the stack grow, as JavaScript does not implement tail call optimization. Hence, while the design is no longer completely stackless, the stack still remains bounded by the maximum nesting depth of expressions in the program.

4.2. asm.js optimizations

Another improvement in performance involved optimizing the code we write and generate in the underlying language, in this case asm.js. One weak point in writing asm.js by hand is that it is designed as a compilation target. Some JavaScript engines therefore assume that common optimizations,

such as the inlining of procedures, are already performed while generating the asm.js code in the first compilation step. This enables faster AOT-compilation of asm.js later on. To compensate for this, our handwritten application requires some profiling to manually identify and optimize certain bottlenecks in performance.

We therefore inline the most common functions in our interpreter by replacing them with macros. Doing so avoids the overhead of function calls by replacing the call with the functions body at compile-time. A macro expander enables us to factor out these function bodies into isolated macros, thereby maintaining the benefits of procedural abstraction. The multi-step expansion of macros in sweet.js also makes it possible to define macros that generate other macros. For instance, the `struct`-macro generates macros for the accessors and mutators of the AST nodes. Such a technique achieves significant performance improvements with a relatively small amount of effort.

Besides inlining common functions on the level of the *interpreter*, it can also be beneficial for performance to inline functions in the *user program*. This is the subject of Section 7.

5. Performance impact of asm.js integration

In order to evaluate performance, the runtimes of a fixed benchmark suite are measured for different versions of the interpreter. These include the four milestones discussed in Section 3.1 (asm0,asm1,asm2,asm3), as well as additional version asm4 that implements the interpreter optimizations described in Section 4. This version can also be compared with the original Slip implementation and the asm.js output that the Emscripten compiler [21] generates from this C implementation. Once we have evaluated the performance of these different versions, we can draw some conclusions that allow us to find a more optimal balance between development effort and performance gains. This will lead to our final implementation, asm5, where some of the components in asm4 are brought back into high-level JavaScript. As discussed in Section 6, doing so can improve maintainability without making any sacrifices in performance with respect to asm4. In fact, we later exploit the improved maintainability to implement more complicated optimizations in the interpreter, which result in increased overall performance (cf. Section 7).

A description of the benchmarks is given below. Most of them originate

from the Larceny R7RS benchmark suite[5].

`tower-fib` A metacircular interpreter is executed on top of another metacircular interpreter. In this environment, a slow recursive Fibonacci is called with input 16. This benchmark also serves as a useful test case, since it uses many features of the interpreter, and therefore provides good coverage.

`nqueens` Backtracking algorithm to solve the n-queens puzzle where $n = 11$.

`qsort` Uses the quicksort algorithm to sort 500000 numbers.

`hanoi` The classical Hanoi puzzle with problem size 25.

`tak` Calculates the Takeuchi function `(tak 35 30 20)` using a recursive definition.

`cpstak` Calculates the same function as `tak`, this time using a continuation-passing style. A good test of tail call optimization and working with closures.

`ctak` This version also calculates the Takeuchi function in a continuation-passing style, but captures the continuation using `call/cc`. It therefore mainly tests the efficiency of this native function.

`destruct` Test of destructive list operations (`set-car!` and `set-cdr!`).

`array1` A Kernighan and Van Wyk benchmark that involves a lot of allocation/initialization and copying of large one-dimensional arrays.

`mbrot` Generates a Mandelbrot set. Mainly a test of floating-point arithmetic.

`primes` Computes all primes smaller than 50000 using a list-based sieve of Eratosthenes.

Each version of the interpreter runs on top of the three major JavaScript VMs found in today's browsers: SpiderMonkey, V8, and JavaScriptCore. SpiderMonkey deserves particular attention here, as it is the only VM implementing AOT-compilation for asm.js and should thus benefit most from

[5] `http://www.larcenists.org/benchmarksAboutR7.html`

asm.js code. The V8 engine found in the Chrome browser does not implement full AOT-compilation for asm.js. Nevertheless, its TurboFan optimizing compiler[6] aims to improve asm.js performance by taking advantage of its low-level style, which opens up many opportunities for optimization. Similarly, JavaScriptCore does not specifically detect asm.js code either. Instead, it takes a more modular approach to optimization by implementing an advanced multi-tier JIT architecture[7]. One of these tiers performs type inference, which works very well for statically typed asm.js code. The final tier uses an LLVM compiler back-end and applies aggressive LLVM optimizations that again work best for low-level code such as asm.js. We therefore expect all engines to benefit in some way from asm.js code [12].

The native C implementation is compiled using Apple's version (6.1) of the LLVM compiler. The test machine is a MacBook Pro (Mid 2012) with a 2.6GHz Intel Quad-Core i7 and 16GB 1600Mhz DDR3 RAM. The VMs were allocated with a 1GB heap to run the benchmarks.

5.1. Integrating asm.js

We first evaluate the performance impact of integrating asm.js into an existing JavaScript application. We compare the benchmark results of `asm0`, `asm1`, `asm2` and `asm3`, representing different milestones in the asm.js integration process (cf. Section 3.1). We slightly modify `asm0` to use the underlying JavaScript memory management for allocation and garbage collection, instead of the memory model described in Section 2.2. This enables a more representative comparison between JavaScript and asm.js, as JavaScript already provides built-in memory management. Figure 3 shows a relative performance speedup for each version in SpiderMonkey, which optimizes the execution of asm.js using AOT-compilation. We obtain these ratios by normalizing all measurements with respect to those of `asm0` and summarize them with their geometric means [8]. These results show that integrating asm.js into the original JavaScript prototype (`asm0`) does not yield the expected performance improvement. In fact, when we only lower down a single component into asm.js (`asm1`), the entire system slows down by a factor greater than 5. On the other hand, the final version with all modules refactored into asm.js does significantly perform better than the original version. In this case, we are seeing a performance improvement of around 80%.

[6] `http://v8project.blogspot.com`

[7] `https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/`

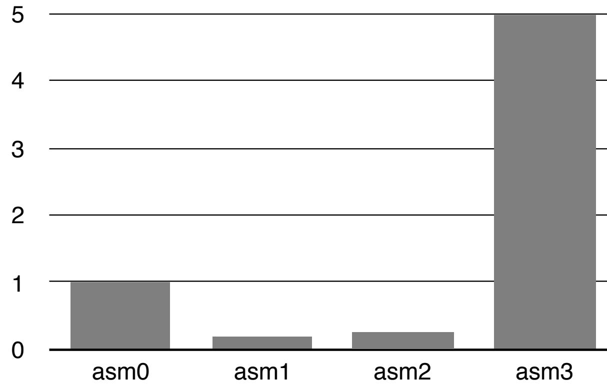

Figure 3: relative speedups in SpiderMonkey (higher is better).

In order to explain these results, we profile each version to determine what causes the initial slowdown. As it turns out, a lot of overhead in SpiderMonkey is caused by calling in and out of asm.js code. This is a known issue with asm.js code that is compiled ahead of time: external JavaScript interfaces asm.js modules through a set of exported functions and vice versa. Passing arguments to those functions requires JavaScript values to be converted and (un)boxed, even between two asm.js modules. Moreover, the function call itself causes trampoline entries and exits in asm.js that build up a severe performance penalty as well. For this reason, it is recommended to contain most of the computation inside a single asm.js module.

For our application, asm1 and asm2 have a lot of calls in and out of asm.js modules, as they refactor the memory model (asm1) and abstract grammar (asm2). Other components rely heavily on these modules, as previously discussed in Section 3.1. In this case, the overhead caused by frequently calling into these asm.js modules is apparently larger than the performance benefits we achieve, hence the slowdown. On the other hand, asm3 uses only a single asm.js module for all the components in the interpreter. Moreover, all major computation resides inside this module. It only calls to external JavaScript for special services (such as I/O) and infrequent tasks (such as parsing and printing). This explains why it does not suffer from the aforementioned overhead and thus greatly benefits from the integration with asm.js.

It is also interesting to examine the performance impact of asm.js on other, non-optimizing engines. After all, we expect asm.js to provide us with general performance improvements, as it claims to be an optimizable sub-

set of JavaScript. Figure 4 shows how JavaScriptCore, an engine that does not perform AOT-compilation for asm.js code, handles the different iterative versions. The results shown are geometric means of relative speedups. In

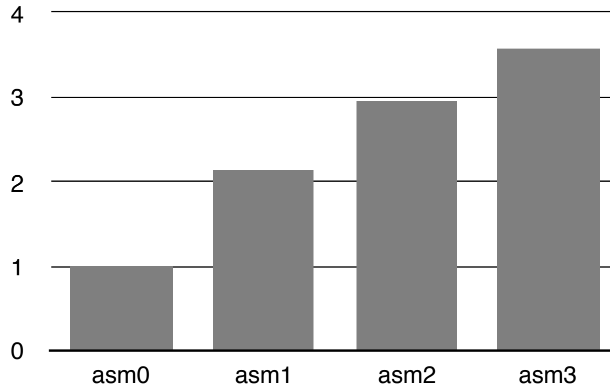

Figure 4: relative speedups in JavaScriptCore (higher is better).

general, we can conclude that the integration of asm.js is beneficial for the other engines in terms of performance. These engines do not compile asm.js ahead-of-time, and therefore do not benefit as much from its presence compared to SpiderMonkey. However, even typical JIT execution of asm.js is able to achieve a significant performance increase over the original version here up to 70%. Moreover, the engine does not suffer from the performance overhead of asm1 and asm2, unlike SpiderMonkey.

5.2. Comparison

We now look at an optimized version of asm3, which we refer to as asm4. Table 1 shows how this version performs in today's most common JavaScript engines. These results demonstrate that the AOT-compilation of asm.js (in SpiderMonkey) is able to provide a significant performance improvement over traditional JIT execution (in JavaScriptCore, V8).

To put these numbers in a wider perspective, we can compare the results from SpiderMonkey with the runtimes of an equivalent native C implementation of Slip. This is a useful comparison to externally validate the results of our strategy. It enables us to determine whether integrating asm.js code into a JavaScript application can bring performance close to native speed. The native C implementation of Slip roughly implements the same level of

	SpiderMonkey	JavaScriptCore	V8
tower-fib	3518	6865	12142
nqueens	1296	2433	4677
qsort	3948	6934	14219
hanoi	4046	8899	18711
tak	878	1629	3374
cpstak	985	2110	4412
ctak	5222	7112	20380
destruct	4350	10029	19643
array1	3518	7724	16161
mbrot	12838	23648	49252
primes	3832	8185	16912

Table 1: runtimes of asm4
(in milliseconds; lower is better).

optimization as asm4, which is described in Section 4. It features the same underlying memory model and garbage collector (cf. Section 2.2), stackless design (cf. Section 2.1), abstract grammar, and module decomposition (cf. Section 3.1) as asm4. Therefore, the native C interpreter operates similarly to our own asm.js implementation.

We can also use the native version for another comparison. We can compile the native C implementation to asm.js using the Emscripten compiler. This represents the typical use-case for asm.js as a compilation target for existing C/C++ applications. Comparing the performance of our handwritten asm.js code to that of the Emscripten compiler is useful for several reasons. Handwriting asm.js code by hand is unconventional, so the performance potential of this strategy is not immediately clear. By comparing the performance of both versions, we can determine whether handwritten asm.js can be efficient in terms of performance compared to what a compiler is able to output. Additionally, it is interesting to see how both strategies compare. One can start with a C application and compile to asm.js using Emscripten, or one can also start with a JavaScript application and gradually integrate asm.js code. The first strategy probably requires more effort for the first step (writing an application in C versus writing it in JavaScript). The second strategy is usually more demanding in terms of producing asm.js code (manual integration vs. using a compiler). In the case of an interpreter, both are plausible strategies to bring a flexible language VM to the web.

We refer to the native C implementation as `native` and to the asm.js code compiled with Emscripten as `compiled`. Figures 5 and 6 illustrate how these versions compare in terms of performance.

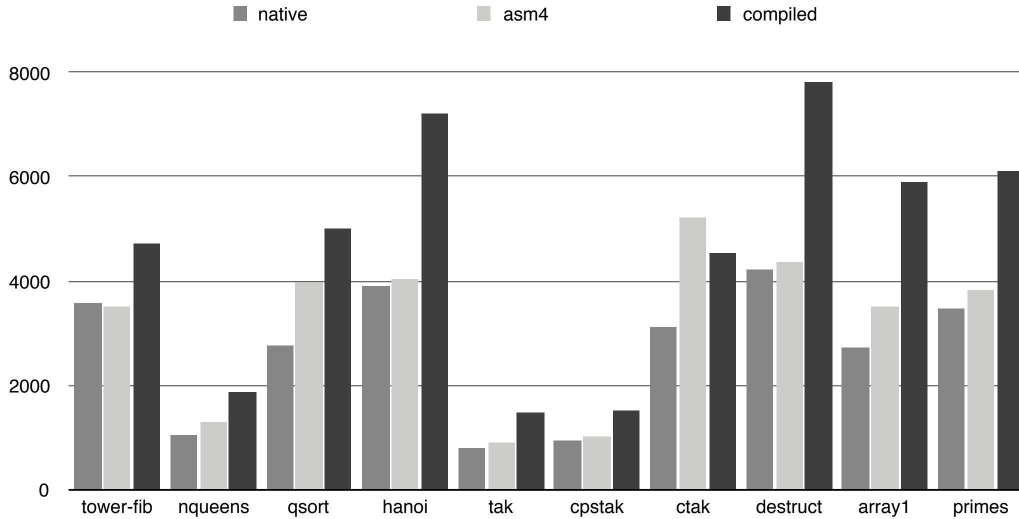

Figure 5: comparison of runtimes using SpiderMonkey
(in milliseconds; lower is better).

Overall, we see that the performance of `asm4` is comparable to that of the native version. We only experienced a slowdown factor of 1.19 in our benchmarks. Typically, asm.js code that is compiled from C/C++ with Emscripten is only twice as slow as the native version [21]. In our experiments, the slowdown factor for the `compiled` version in SpiderMonkey was 1.74. This makes it around 46% slower than our handwritten implementation.

However, we have to interpret these results carefully. While `asm4` is very similar to the C implementation, they are not completely identical. For instance, there is difference in how both interpreters represent the stack. The native implementation uses a linked-list representation in the heap, while `asm4` stores the stack contiguously in memory. One case in which this difference becomes obvious is the `ctak` benchmark, which tests the performance of `call-with-current-continuation`. To construct the current continuation in `asm4`, the entire stack must be copied into a separate data structure, while the linked-list implementation only needs to copy the pointer to the current stack. As a result, `native` and `compiled` perform significantly better on this benchmark compared to `asm4`. On the other hand, using a contiguous

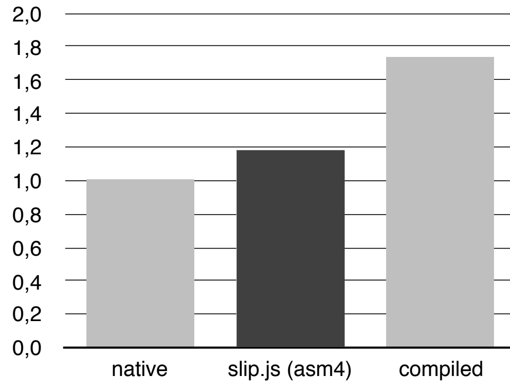

Figure 6: slowdown factor over native using SpiderMonkey (lower is better).

stack might be beneficial for general performance due to better cache rates and automatically reclaimed memory from stack frames (causing less garbage collections). The C implementation tries to avoid allocation of fresh stack frames by deallocating stack frames into a pool and reusing these pooled frames whenever possible. In general, these subtle differences make it hard to make a rigorous performance comparison between both versions. Therefore these results should not be interpreted as a net performance gain or loss over the native implementation. Instead, the comparison is useful to show that both implementations have similar performance characteristics. One implementation does not significantly outperform the other, and it is clear that asm.js brings the performance of our interpreter much closer to native speeds. These particular results only show a small sacrifice in performance for `asm4`, which is acceptable considering that it is immediately available on web platforms and its many users.

In the same way, we can also compare `asm4` to the `compiled` version. Again, we cannot draw precise conclusions here. Instead, it appears that both strategies can result in asm.js applications that are close to native speed. The preferred strategy therefore most likely depends on the development effort, which we previously discussed. The availability of an existing C or JavaScript implementation could be a decisive factor. However, compilation to asm.js is not always straightforward, as Emscripten requires the original source code to be "portable". For instance, the C implementation of Slip is not completely portable in this sense due to unaligned memory reads and writes. This resulted in some problems when working with floats

in SpiderMonkey, which is why we have omitted the `mbrot` benchmark here. Similarly, JavaScriptCore was unable to run the compiled version unless we lowered the LLVM optimization level (which in turn degraded performance). Emscripten specifies the behavior of unportable C code as undefined. Moreover, we argue that our approach is more flexible, because one can start with a straightforward JavaScript application, and incrementally integrate as much asm.js as necessary to achieve acceptable performance.

6. Revisiting asm.js integration

In our initial integration described in Section 3.1, we refactored most of the modules into asm.js. Doing so allows us to evaluate the development effort and performance gains of integrating asm.js into an existing JavaScript application. In addition, by directly implementing the optimization of `asm4` in asm.js we are able to assess the maintainability of handwritten asm.js code. It is clear that while asm.js provides significant performance improvements, it also introduces an extra development effort, both in the short term when writing components in low-level asm.js as well when evolving these components in the long term. Hence, a better solution is to lower down only the most critical components in order to improve maintainability and reduce development effort, while still maintaining the performance benefits of asm.js

In this Section, we rewrite asm.js components that are not performance-critical into plain JavaScript, resulting in version `asm5` of our interpreter. We note that the most critical components in an interpreter are those used heavily during evaluation: the evaluator itself, the abstract grammar, the memory management and the natives. Hence their implementation in asm.js yields the significant performance improvements, as we measured in Section 5. On the other hand, the compiler, the printer, the pool and the dictionary are only used once before (or after) evaluation[8]. Hence for these components we can sacrifice performance for development effort and maintainability, without impacting the overall performance of our interpreter (which is measured by its evaluation time). As such, by implementing components such as the compiler in high-level JavaScript it becomes easier to further extend the interpreter with more complex (compile-time) optimizations. We demonstrate this in the next section by adding an important optimization, function inlining, to our improved high-level implementation of the compiler.

[8]except in the presence of calls to meta-level natives such as `eval`, `read` or `display`.

Finally, note that it is not just possible to reuse the JavaScript implementations of these components from `asm2`. While these were implemented in plain JavaScript, they already were prototyped in a low-level style, using registers and a continuation-passing style with a trampoline to allow for a more direct translation to asm.js later on. Furthermore, to facilitate the memory management and interoperability with other components, the decision was made to lower almost all components down to asm.js in `asm3`. However, now that we implement some components in high-level JavaScript, we encounter several challenges to ensure correct interoperability between JavaScript components and the asm.js environment. More precisely, we discuss how *handles* are introduced to avoid a new memory management problem that occurs when refactoring components back to JavaScript.

6.1. Using handles to avoid corrupt references

While JavaScript provides automatic memory management, asm.js does not have a built-in garbage collector (GC). Instead, an asm.js module has access to one large `ArrayBuffer` that serves as the heap and is allocated from the JavaScript host code when initializing an asm.js module. Without a built-in garbage collector, code written in asm.js has to manage this memory heap itself. Therefore our interpreter implements a mark-and-sweep garbage collection algorithm to free unused objects (chunks) from the memory heap. As previously discussed, our interpreter can easily collect the 'root set' of active references by iterating over the Slip stack and registers. Our mark-and-sweep garbage collector includes a compacting phase, so that after a garbage collect all 'active' memory is contained in one contiguous block, followed by another contiguous block of free memory. This avoids gaps of free memory between active chunks, but also implies that chunks may move to another location in the heap, causing corrupted memory references if not all references are updated to point to their new memory location after a GC. In our asm.js module, however, references on the stack and in the registers are easy to update.

There is a problem when plain JavaScript code refers to objects residing in the asm.js memory heap. Consider Figure 7, in which a JavaScript variable `exp` contains a reference to some object inside the memory array managed by our asm.js interpreter. At some point during evaluation, a garbage collection is triggered and during the compaction phase our collector relocates objects in the asm.js heap. Pointers in the asm.js world are managed by the interpreter and are correctly updated to point to the new location of their values.

However, because variable `exp` is not under control of the asm.js interpreter (it is not on the stack, in a register, or even inside the asm.js module), our memory manager can not update the reference after GC. Consequently, it may now point to an incorrect location, making it a corrupt reference. The resulting situation is shown in Figure 8. Note that the JavaScript variable is also not included in the root set of active references. This means that if no other reference in the asm.js module were to exist to that same chunk, it would have been freed incorrectly by our GC algorithm.

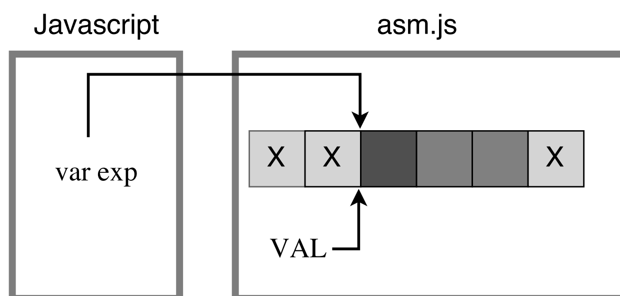

Figure 7: Cohabitation of JavaScript and asm.js, `exp` is a JavaScript object pointing to the same memory location as `VAL`.

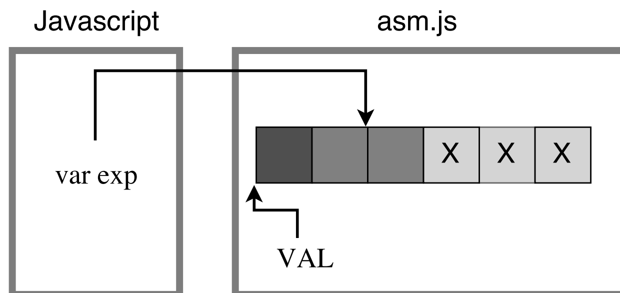

Figure 8: Cohabitation of JavaScript and asm.js after compaction, the `VAL` pointer has been modified by the GC, but `exp` points to the wrong location.

To resolve the problem of corrupted references in JavaScript code due to our compacting collector, we resort to *handles*. Instead of pointing directly into asm.js memory, references in JavaScript now point into a *handle memory* that resides in asm.js, as illustrated in Figure 9. When compaction happens, the handle memory is updated by the garbage collector in the asm.js module so that all its pointers point to the correct memory location. Because the JavaScript index into the handle memory does not change, each JavaScript

object still points (indirectly) to the correct memory location, as shown in Figure 10. Allocating and releasing handles in the handle memory is managed by a free list. To ensure proper interaction between JavaScript and asm.js modules, raw references are not allowed to escape from asm.js into JavaScript. Instead, we enforce that all escaped references are first converted to handles, so that corrupted references in JavaScript are no longer possible. As a result, before calling JavaScript functions from asm.js, all arguments that reference objects in the asm.js heap are first converted to handles by extending the handle memory with a new handle pointing to the argument reference. Similarly, besides primitive datatypes JavaScript functions can only return handles. Therefore, after calling a JavaScript function, any reference return value is dereferenced in the handle memory. In the opposite direction, asm.js functions that can be called from JavaScript return handles instead of a raw memory references. asm.js functions also have to dereference any handles they might be passed as arguments from the outer JavaScript environment.

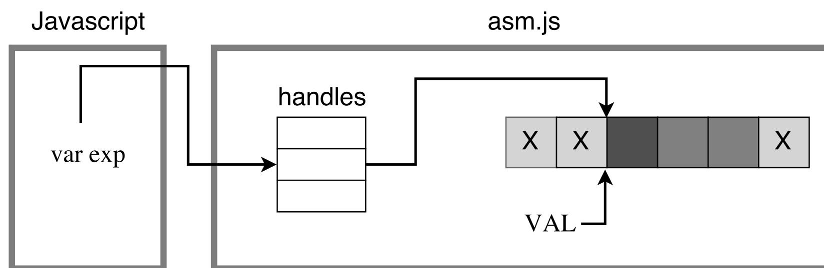

Figure 9: Cohabitation of JavaScript and asm.js with the use of handles, before compaction.

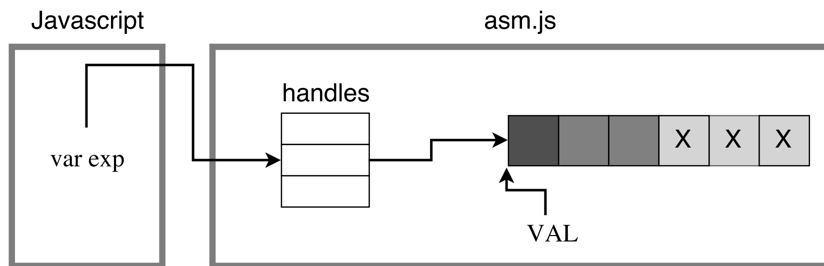

Figure 10: Cohabitation of JavaScript and asm.js with the use of handles, after compaction. The JavaScript variable `exp` did not need to be updated and still points to the correct location.

The size of the handle memory has to be managed as well, otherwise it would keep growing as new handles are created and would eventually consume all available memory. Most of the handles result from the local variables of a function (stored on the JavaScript stack), and should be freed on function exit, when they are no longer needed. Doing this manually would be tedious, especially in the presence of exceptions that can be thrown during the execution of the function. However, this process cannot be automated easily, as it would require performing reflective operations to retrieve the local variables declared with `var` in a function, which is not possible in JavaScript. We therefore opted for a semi-automated solution. When asm.js values are used in JavaScript, we do not bind them to `var`s, but we attach the handles explicitly to a JavaScript handle object, which is initially empty on function entry and bound to the `this` keyword. At function exit, all handles that are attached to the handle object can be freed. To automate the process, we extend the `Function` prototype with a *lifting* function, so that a call to a lifted function first creates an empty handle object, then calls the original function with `this` bound to the handle object, and after this call frees all the handles that were attached to the handle object. In order for this strategy to work, JavaScript functions should adhere to certain rules. Only handles whose lifetime is bound to the dynamic extent of the function should be bound to the `this` object as previously described. Using such handles after a function has returned (for instance by storing them in a global data structure) is not allowed, because these handles will have been automatically freed on function exit. If a handle has to be available outside of the dynamic extent of a function, it should not be attached to the `this` object, so that it is explicitly protected from being automatically freed. Only handles can be attached to `this`, other local variables should just use `var`s. Note that there is no particular reason why we chose to use `this` other than convention. As an alternative, we could have passed the handle object as an extra argument to the function. While this solution can be expressed natively in JavaScript using prototypes, one can arguably automate and separate this concern better using an aspect-oriented library for JavaScript. Alternatively, this mechanism could also be automatically generated using some preprocessor or macro expander.

6.2. Impact on maintainability

Using handles we are able to remove asm.js code from non-critical components, thereby improving readability and maintainability. To illustrate this,

Table 2 contains an excerpt of the compilation of conditionals written in asm.js with the use of macros (in `asm4`, on the left), and in JavaScript with the use of handles (in `asm5`, on the right). The JavaScript version is more readable, more maintainable and less prone to bugs. This is because it is written at a higher level, avoiding the low-level characteristics of asm.js. For instance, in asm.js we are forced to explicitly manage a stack for the garbage collection, and use a continuation-passing style to simulate exceptional control flow. Instead, we now get access to the high-level features of JavaScript, such as first-class functions, exception management and JavaScript objects.

In addition to the improved readability, we compare the code size of the components that were rewritten in plain JavaScript in Table 3.

We observe that especially the compiler gains from the refactoring, as its size is almost halved in terms of lines of code. However, the total code size is not significantly reduced, as it goes from 6804 lines of code in `asm4` to 6631 lines of code in `asm5`. This is because of two additions that were required to make the refactoring possible: the management of the handle memory (110 LOCs), and the technique of storing handles in the `this` object (40 LOCs). However, these additions are a one-time effort and would not further increase the code size when further extending other components. If, for example, the compiler is further extended with a new optimization (in Section 7), both the effort as well as the additional LOCs would be smaller. The overall code size also slightly increased because the code was split into multiple files.

Overall, bringing non-critical components back to JavaScript allows us to reduce the complexity of our implementation.

6.3. Impact on Performance

We previously observed significant performance increases when integrating asm.js (cf. Section 5). We therefore can expect the opposite to happen when moving from asm.js back to plain JavaScript, especially when this refactoring introduces handles and pointer indirection. This indirection has a negative impact on performance because an extra dereference step is required every time an object is accessed in memory.

However, although the refactored components in Table 3 run slower than their asm.js originals, we observed that this did not incur any significant loss of performance when evaluating Slip programs. The reason is that the refactored components are only called during the compilation phase. The compilation phase is typically only a small fragment of the running time of

<pre>C_compileIf {
  if(!(isPair(LST)|0)) {
    err_invalidIf();
    goto error;
  }
 	
  EXP = pairCar(LST)|0;
  LST = pairCdr(LST)|0;
 	
  if(!(isPair(LST)|0)) {
    err_invalidIf();
    goto error;
  }
 	
  claim()
  push(KON);
  push(LST);
  push(TLC);
  TLC = __FALSE__;
  KON = C_c1_if;
  goto C_compile;
}
 	
C_c1_if {
  TLC = pop()|0;
  LST = peek()|0;
  EXP = pairCar(LST)|0;
  LST = pairCdr(LST)|0;
  poke(VAL);
 	
  if(isNull(LST)|0) {
    ...
  }
 	
  if(isPair(LST)|0) {
    ...
  }
 	
  err_invalidIf();
  goto error;
}</pre>
	<pre>function compileIf(exp,tailc,inline) {
	  if(!isPair(exp))
	    compilationError(err.invalidIf);
	
	  this.predicate = car(exp);
	  this.branches = cdr(exp);
	
	  if(!isPair(this.branches))
	    compilationError(err.invalidIf);
	
	  this.alternative = cdr(this.branches);
	
	  if(asm.fisNull(this.alternative))
	    ...
	  else if (asm.fisPair(this.alternative)) {
	    ...
	  } else
	    compilationError(err.invalidIf);
	}</pre>

Table 2: Compiling conditionals, in asm.js with macros (asm4, left), and in JavaScript (asm5, right).

Component	asm4	asm5
Compiler	637	368
Parser	236	227
Dictionary	69	67
Symbol pool	79	22

Table 3: Physical LOCs of components that are translated from asm.js to plain JavaScript.

a program, which is dominated by evaluation of the rich abstract grammar items generated by the compiler.

To verify this claim, we ran the benchmarks of Section 5 on `asm4` and `asm5`. and observed that running times were similar in both versions (varying from a slowdown of 2% to a speedup of 4%). One exception is the `tower-fib` benchmark, which makes heavy use of `eval`, requiring the compiler to be called multiple times at runtime and therefore resulting in a 35% slowdown at worst. In general, programs that use meta-level natives such as `eval` and `load` invoke the refactored parser and compiler, and will therefore run slower. However, in most conventional programs the usage of such natives is limited or non-existent.

This experiment demonstrates the philosophy and strength of our approach. We use asm.js for the core components of our JavaScript application, in this case the run-time infrastructure of the interpreter. This ensures that overall performance goals for the interpreter are met, as confirmed by the running times of our benchmarks. For the components that do not significantly impact overall performance, we use plain JavaScript instead of asm.js to improve maintainability and reduce development effort. This negatively impacts the individual performance of these components. The refactored parser and compiler run significantly slower than their previous versions, as we already observed in the `tower-fib` benchmark. Stress-testing the parser revealed that the JavaScript version is two orders of magnitude slower than its previous asm.js implementation. However, because the parser generally does not play a significant part in the overall evaluation process, we consider the use of JavaScript for the parser and compiler beneficial.

7. Adding function inlining

With the non-critical components written in plain JavaScript, it becomes less of a burden to experiment and extend the interpreter. We can for exam-

ple add more advanced compiler optimizations to the high-level JavaScript implementation of the compiler.

In this section, we investigate adding *function inlining* as an optimization offered by the interpreter. Function inlining consists of replacing function calls by copies of the function body in the user program, with the goal of increasing performance by avoiding the overhead of function calls at the expense of increasing the code size [16]. We chose inlining as it is considered one of the most valuable optimizations, is not trivial to implement and has already been explored for Scheme-like languages in related work [19, 7].

7.1. Inlining in practice

We recall that when the interpreter evaluates an expression, it first parses this expression into an AST, and then compiles this AST into an enriched grammar that actually gets evaluated. Inlining happens at compile-time, when compiling the AST into an enriched grammar term.

We opt for an annotation-driven approach to inlining. A function definition can be annotated by using `define-inline` instead of `define`. The compiler will try to inline functions defined through `define-inline` if it fulfills certain requirements (see Section 7.2). This implies that annotated functions are only eligible for inlining, but the final decision is always up to the compiler. On the other hand, annotating functions with `define-inline` is always safe for the user. Potential inlining problems, such as variable and parameter shadowing, are all taken care of by the compiler and if inlining is not possible, the annotation will simply be ignored. However, functions defined using `define-inline` can not be reassigned, as doing so would require replacing the function bodies that are already inlined. Because of the presence of `eval` and the interactive nature of Slip's *read-eval-print*-loop, it is not possible to perform whole-program analysis and detect such assignments before the inlining decision is made. Hence, when attempting to reassign an inlined function, an error will be thrown to the user. Another solution to this problem would be to implement speculative inlining [7], which is beyond the scope of this paper.

7.2. Implementation

An important performance benefit from inlining comes from the fact that arguments at the call site can usually be directly substituted in the function body. However, one needs to be careful about this, as this generally

only applies to simple arguments, such as numbers, variables and other literal values. As an example, consider a program with the following call to `sum-of-squares`, where `f` is a function that performs some complex operation and possibly includes side effects.

```
(define-inline (square x)
  (* x x))
(define-inline (sum-of-squares x y)
  (+ (square x) (square x)))

(sum-of-squares x (f x))
```

Naive inlining results in a program that is not equivalent to the original:

```
(+ (* x x) (* (f x) (f x)))
```

In this version of the program, the work done by `f` and its potential side effects are duplicated, which breaks correctness. To remedy this situation, we require that complex expressions in argument positions have to be computed once before being inlined. The resulting code to evaluate would be conceptually equivalent to the following.

```
(let (($y (f x)))
  (+ (* x x)
     (* $y $y)))
```

In practice, we do not introduce `let`-bindings as it requires introducing new lexical frames during evaluation. Doing so would cause an extra overhead, as certain design choices in our implementation make it somewhat expensive to capture and extend the current environment. Instead, we introduce a new abstract grammar item, which we internally call a *bind*. This binding operation more closely resembles a traditional `define`, which is also the preferred form of binding in Slip. The main difference is that variables introduced with these bindings are treated as temporaries, which are only scoped to the inlined body and whose memory locations can be reused after the function has been inlined.

Another problematic situation is the one of recursive functions [19]. When recursive functions are inlined, the compiler has to be careful and bound the inlining, otherwise it would keep replacing a recursive function call by its body indefinitely. However, it can be beneficial to inline a function for several iterations, as doing so has an effect similar to *loop unrolling* [19]. Because of this, the user can specify a global parameter for the maximum allowed *inlining depth*, which specifies how deep a particular function can be

inlined. Setting the depth to zero disables inlining, while setting it to one causes only simple functions to be inlined, but not recursive ones. Setting the depth to values higher than one will also enable inlining for recursive functions up to a given depth in order to achieve the effect of loop unrolling. Note that recursive functions are only inlined within their own body (up to the specified inlining depth), but not at the external call sites.

Even though the user has to annotate functions that have to be inlined with `define-inline`, this is only an *advice* to the compiler, which takes the final decision of whether or not to inline a function. The compiler maintains certain restrictions on inlining, based both on the inlined function and its call site. The reasons behind these restrictions are twofold. On one hand, we want to limit the inlining to small functions, as otherwise the function call overhead becomes proportionally small and the increase in code size can negatively impact performance [19]. While our restrictions are not directly size-based, larger functions are more likely to violate one of them. On the other hand, we exclude certain rare situations that would make the implementation either too complex or inefficient when taken into account. For instance, the compiler does not inline functions that themselves contain nested function definitions. Such functions are typically too large to benefit from inlining, and nested function definitions extend the lexical scope of the function that is being inlined. By enforcing certain restrictions in our implementation, the compiler is allowed to make certain assumptions about the current execution frame while inlining a function body, simplifying the implementation of inlining.

When the compiler performs inlining, it checks that the requirements are met. The decision on whether a candidate function for inlining is actually inlined or not happens when compiling the function *call*, not when compiling the function *definition*. If any of the assertions fail because a restriction is violated, an exception is thrown and the interpreter falls back to regular compilation of the function call. When this happens, that function is marked and is no longer considered for inlining later on at other call sites. Note that exceptional control flow is not available in asm.js, and is one of the reasons we were previously forced to employ a continuation-passing style. Its availability in JavaScript to help us implement inlining is one of the benefits from having the compiler written in a high-level language.

7.3. Validation

We validate our implementation by running specific benchmarks that may profit from inlining. In Table 4, we compare the results of running bench-

marks with inlining disabled (**no inlining**, i.e. `define-inline` is equivalent to a regular `define`) and with inlining enabled (**inlining**), where a maximum inlining depth of 4 is used.

	no inlining	**inlining**	**speedup**
`sum-of-squares`	3066	2114	31%
`matrix-multiplication`	8777	4522	48%
`loop`	12852	12119	6%
`fib`	6033	5447	10%
`hanoi`	4026	3144	21%

Table 4: runtimes of `asm5` with and without inlining
(maximum inlining depth set to 4, times are in in milliseconds; lower is better).

We chose the following benchmarks because of the opportunities they contain for applying function inlining:

- **sum-of-squares**: the `sum-of-squares` function, called 10^7 times.
- **matrix-multiplication**: multiplication of two-dimensional matrices, defined on top of multiple layered data abstractions (called 10^6 times).

The call to `sum-of-squares` can be executed more efficiently at run-time, because it is reduced to a simpler expression at compile-time once the user-defined function calls have been inlined. In total, the evaluator can therefore avoid three extra function calls, which results in a significant speedup. The same argument also holds for the second benchmark. However, in this case the speedup is even larger. This is because matrix multiplication is defined on top of multiple abstractions whose overhead can all be avoided with inlining. In fact, the matrix multiplication at the top layer can be completely reduced to a large expression that does not include any function call to a user-defined procedure defined in one of the layers.

Additionally, we also included benchmarks to assess the performance impact of unrolling recursive functions with inlining.

- **loop**: a tail-recursive function computing Fibonacci numbers, called once on the number 10^8.
- **fib**: a tree-recursive function computing Fibonacci numbers, called once on the number 36.

- **hanoi**: another tree-recursive function, solving the Hanoi towers problem with input 25. This benchmark is also in the set of performance benchmarks we used in Section 5.

Note that loop unrolling has a larger effect on tree-recursive functions here, because the inlining depth has an exponential loop unrolling effect on them (e.g. with an inlining depth of 4 the tree-recursive Fibonacci function actually gets inlined 2^4 times). This of course also dramatically increases their code size. The reason why the `hanoi` benchmark has a better speedup is because it is almost a best-case scenario for the inlining: 3 out of 4 arguments passed to the recursive calls are simple arguments that can be directly substituted in the bodies of the next iterations. In the `fib` benchmark, this is not possible, because the only argument that is passed in the recursive calls in a complex one that still has to be computed and bound at run-time.

8. Related and future work

There are a number of approaches to implement dynamic languages on the web [21] . However, most of these approaches focus on porting existing applications to the web instead of providing near-native performance for full language implementations in the browser. For instance, one common approach to support a new language is plain compilation to JavaScript. Several examples of such implementations exist[9]. However, as the main focus in these implementations is portability rather than performance, they usually target high-level JavaScript instead of asm.js. As previously discussed, a more straightforward approach is to write an interpreter for a language in JavaScript. Several examples of such interactive programming environments can be found online[10]. They allow easy experimentation with new languages in the browser, but usually the obtained performance is significantly worse than the performance of existing native implementations for these languages [6]. Emscripten [21] enables an implementation approach with more focus on performance by compiling an existing VM written in C/C++ to asm.js. Emscripten provides a compiler back-end to generate asm.js from the LLVM intermediate representation format [15]. Any language that compiles into the LLVM IR can therefore be compiled into asm.js. Existing

[9] `http://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS`

[10] `http://repl.it`

language implementations such as LuaVM and CPython have already been ported to the web using this strategy. In Section 5 we discuss this strategy when describing the compilation of the original C implementation of Slip to asm.js using the Emscripten toolchain.

PyPy.js [14] provides a Python interpreter for the web. It relies on PyPy [3] which implements Python in a restricted subset of Python that compiles to C. The C code can subsequently be compiled to asm.js via Emscripten. This is a different strategy from ours, since the asm.js code is generated and not written by hand as in slip.js. PyPy.js also features a customized JIT back-end which emits and executes asm.js code at runtime.

LLJS [17] is a low-level dialect of JavaScript that was initially designed to experiment with low-level features and static typing in JavaScript, but it never reached a mature and stable release. While our approach uses domain-specific macros that translate directly into asm.js, LLJS defines a more general-purpose, low-level language that can also translate into asm.js.

WebAssembly [10] (wasm) is an open standard for a portable and efficient code format suitable for compilation to the web. wasm is a next step in bringing native performance to the web, and resolves the shortcomings of using asm.js for this purpose. Advancing asm.js would mean that JavaScript increasingly has to be extended with low-level operations and data structures for it to serve as a compilation target (asm.js being a subset of the language), turning it into a bottleneck. By moving to a new format that is not a subset of JavaScript, wasm is able to overcome these difficulties. WebAssembly no longer has to make any compromises to stay backward compatible with JavaScript, making it easier to add new features that are required to reach native levels of performance. However, because asm.js and wasm have a common low-level nature, tool infrastructure (compilers) can reuse and share existing infrastructure for both formats [2]. Like asm.js, wasm is not really regarded as a programming language, but as a format that is meant to be generated and manipulated by tools [2]. However, next to a binary format, wasm also has one or more textual formats that are in the process of being defined and proposed, so it will be possible to directly program in wasm.

We introduced inlining in our interpreter to assess the effort required to add complex optimizations. We therefore did not focus on further maturing the inlining itself, although related work shows useful improvements that could be included. For instance, [19] presents an algorithm to make better decisions on when to inline to avoid uncontrolled code size growth. Inlining is presented not only as an optimization useful to eliminate function call

overheads, but also as a means to enable other compiler optimizations (such as constant propagation, constant folding and dead-code elimination) across function boundaries. We avoided the problem of reassigning inlined functions by simply throwing an error when the user attempts to do so. In [7], a speculative approach is taken to deal with this issue. Interestingly, the paper does not focus on inlining user-defined, but rather native functions. Indeed, applying optimizations to native functions would be interesting to combine with our existing inlining of user-defined functions to improve performance even further. Finally, we refer to [13] as an excellent general reference for many of the aspects of inlining in functional languages.

9. Discussion

The idea of embedding low-level code into another (higher-level) language to improve performance is not new. For example, C applications can contain inlined assembly code, and many high-level programming languages have a native interface to call performance-critical procedures written in a lower-level language like C. This kind of approach also lies at the core of our own asm.js integration strategy: we start with a high-level JavaScript application and rewrite critical sections of the code in asm.js that is called from JavaScript to improve the overall performance.

Our experiment however reveals several limitations to this approach in our particular setting: the high overhead of calling in and out of asm.js code prevents us to apply fine-grained optimizations[11]. As discussed, performance benefits are only realized when large computations are contained in asm.js, forcing the use of coarse-grained and isolated asm.js modules. In the case of our interpreter, we had to refactor the entire run-time infrastructure to asm.js to avoid calling back into JavaScript during evaluation. It would be preferable to only lower down smaller but critical pieces of code one at a time, such as the implementation of certain frequently used natives in our language (e.g. `+` or `cons`). In general, one would then be able to start with a straightforward interpreter for a dynamic language, and afterwards gradually refactor the code to asm.js in small steps, using detailed and fine-grained profiling information to determine the next part(s) of code to be lowered down, which was the initial goal of our experiment.

[11] a 'flaw' in asm.js, since it did not envision this use-case in its design.

Fortunately, things are improving with the arrival of WebAssembly, which, at the time of writing is still in its development stage.As a successor to asm.js, WebAssembly currently has laid out several goals [10] that are very appealing to our approach. For instance, it aims to provide a human-editable textual format for the language, as well as a binary representation. One of the design goals of asm.js was to remain backward compatible with JavaScript, resulting in syntactic overhead. WebAssembly's new textual format is designed to allow more natural reading and writing, for instance by dropping all the type coercions that were previously required for asm.js validation. In doing so, WebAssembly claims to open up more possibilities for testing, optimization, and experimentation. For our integration strategy, this should make it much easier to write WebAssembly code by hand. It would be interesting to examine to what extent the improved textual format could eliminate the need to use macros, which are currently a necessary tool for our approach. WebAssembly also provides a better programming model, for instance by supporting (zero-cost) exceptions. Another benefit is that WebAssembly facilitates integration with JavaScript code. For instance, unlike asm.js, wasm code is able to access garbage-collected JavaScript objects and integrates into the ES6 module system. Furthermore, it seems likely that the overhead of calling in and out of JavaScript is lower compared to asm.js, allowing the application of our integration strategy at a finer level of granularity.

With more focus on integration with JavaScript and a human-editable text format, WebAssembly becomes more appealing than asm.js in our proposed integration strategy for improving the performance of existing JavaScript applications. Given these recent developments, it would be interesting to repeat our experiments using WebAssembly instead of asm.js. The current design choices promise to improve on many shortcomings that we encountered, mainly in terms of development effort. We believe that, once stable and mature, WebAssembly is a better fit for our approach than asm.js.

10. Conclusion

Overall, our experiments allow us to evaluate the impact of integrating asm.js and writing asm.js applications by hand in general. In terms of performance, our strategy of integrating asm.js into a JavaScript application yields considerable improvements, as we achieve near-native performance on the web. Additionally, we can make the following conclusions on the usage of asm.js:

- Using asm.js to improve the efficiency of web applications comes down to a tradeoff between development effort and performance. We concluded that the optimal strategy is to limit the usage of asm.js only to the performance-critical components of a JavaScript application. This way, one can preserve maintainability and reduce development effort, while still maintaining the performance benefits of asm.js.
- Frequently calling in and out of asm.js modules compiled ahead-of-time causes a major overhead in terms of performance. Integrating asm.js into an existing JavaScript application is therefore only beneficial if all computation can reside in a single asm.js module.
- A macro preprocessor is necessary to alleviate the challenges in readability, maintainability and performance when writing asm.js by hand.
- In the case of interpreters, our integration strategy yields considerable performance improvements that result in an efficient language implementation on the web. Furthermore, by implementing our compile-time optimizations in high-level JavaScript, we can easily implement additional advanced optimizations to further improve the run-time performance of the interpreter.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 2 edition, 1996.

[2] V. M. S. S. . Ben L. Titzer. A Little on V8 and WebAssembly. `https://ia601503.us.archive.org/32/items/vmss16/titzer.pdf`. [Accessed: 23/06/2016].

[3] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In I. Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, pages 18–25. ACM, 2009.

[4] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[5] T. D'Hondt. A brief description of Slip. https://github.com/noahvanes/slip.js/raw/master/varia/Slip.pdf, 2014. [Accessed: 18/09/2015].

[6] L. Domoszlai, E. Bruel, and J. M. Jansen. Implementing a non-strict purely functional language in javascript. *Acta Universitatis Sapientiae*, 3:76–98, 2011.

[7] M. Feeley. Speculative inlining of predefined procedures in an R5RS scheme to C compiler. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 237–253, 2007.

[8] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, 1986.

[9] D. P. Friedman and M. Wand. *Essentials of programming languages (3. ed.)*. MIT Press, 3 edition, 2008.

[10] W. W. C. Group. WebAssembly. http://webassembly.org. [Accessed: 13/12/2016].

[11] B. Hackett and S.-y. Guo. Fast and Precise Hybrid Type Inference for JavaScript. *SIGPLAN Not.*, 47(6):239–250, June 2012.

[12] D. Herman, L. Wagner, and A. Zakai. asm.js specification. http://asmjs.org. [Accessed: 04/08/2015].

[13] S. L. P. Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(4&5):393–433, 2002.

[14] R. Kelley. PyPy.js. http://pypyjs.org. [Accessed: 02/12/2015].

[15] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[16] R. Leupers and P. Marwedel. Function inlining under code size constraints for embedded processors. In J. K. White and E. Sentovich, editors, *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design, 1999, San Jose, California, USA, November 7-11, 1999*, pages 253–256. IEEE Computer Society, 1999.

[17] Mozilla. LLJS: Low-Level JavaScript. `http://lljs.org`. [Accessed: 11/08/2015].

[18] C. Queinnec. *Lisp in small pieces*. Cambridge University Press, 2003.

[19] M. Serrano. Inline expansion: When and how? In *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Track on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, pages 143–157, 1997.

[20] N. Van Es, J. Nicolay, Q. Stievenart, T. D'Hondt, and C. De Roover. A performant scheme interpreter in asm.js. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1944–1951, 2016.

[21] A. Zakai. Emscripten: an llvm-to-javascript compiler. In C. V. Lopes and K. Fisher, editors, *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 301–312. ACM, 2011.