# A Portable VM-based implementation Platform for non-strict Functional Programming Languages

Jan Martin Jansen
Faculty of Military Sciences,
Netherlands Defence Academy
P.O. Box 10000, 1780 CA Den Helder, the Netherlands
jm.jansen.04@mindef.nl

John van Groningen
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
johnvg@cs.ru.nl

## ABSTRACT

The Web has become a paramount deployment platform for computer applications. Modern Web applications require execution of code on both server- and client-side. For the client-side JavaScript is the more-or-less default platform for execution of code. Users of Functional Programming languages like Haskell and Clean, who want to develop their software completely in these languages, are forced to rely on dedicated methods to transform their code to something that can be executed on top of JavaScript. In this paper we introduce a Virtual Machine that is capable of efficient execution of bytecode generated by a compiler for a non-strict intermediate functional language. The virtual machine has several implementations supporting the same bytecode, including JavaScript and asm.js versions. In this way we obtain a portable execution platform for non-strict Functional languages with a better client-side performance than existing client-side execution platforms.

## CCS CONCEPTS

• **Information systems** → **Web applications**; • **Software and its engineering** → **Virtual machines**; **Functional languages**; **Interpreters**; *Compilers*;

## KEYWORDS

Functional Programming, Virtual Machines, Bytecode, Haskell, Clean

**ACM Reference format:**
Jan Martin Jansen and John van Groningen. 2017. A Portable VM-based implementation Platform for non-strict Functional Programming Languages. In *Proceedings of IFL Conference, Leuven, Belgium, August 31-September 2, 2016 (IFL 2016),* 14 pages.
DOI: http://dx.doi.org/10.1145/3064899.3064903

## 1 INTRODUCTION

Web applications require execution of client-side code mostly to obtain decent performance, especially for the more interactive parts of these applications. For example, the latest version of the iTask system [1, 21, 23] depends heavily on client-side executed JavaScript for the rendering of SVG-images.

At this moment JavaScript is the standard platform for client-side processing and is included in every major web-browser.

The greatest advantage of using JavaScript is that it works out of the box (the code can be simply added as a script to a web-page) and does not require the loading of additional plug-ins. Although traditionally perceived as being much slower than languages such as Java, C(++) and functional languages like Haskell and Clean, the introduction of JIT compilers for JavaScript and the use of dedicated subsets of JavaScript like asm.js [12], has changed this picture significantly. Using these techniques, client-side code can be executed at a speed that comes close to that of native C(++) code (1.5 - 2 times slower [12]).

For an application programmer used to Functional languages like Haskell or Clean writing large parts of the application in JavaScript is an unpleasant task. A remedy for this is to write the entire web-application in Haskell or Clean and to use dedicated tools for generating the parts that should be executed at the client-side. For this several solutions exist. One solution is to use dedicated plugins for the execution (interpretation) of client-side code. This is done, for example, for the Sapl interpreter, a client-side platform for Clean [13, 22] using a JavaApplet plugin. But the solution of the installation of a plug-in, is often infeasible in environments where the user has no control over the configuration of his/her system.

Another solution is to compile Functional code to JavaScript that be can executed directly at the client-side. This is the current solution for the iTask-system [2, 21], for Haste [5, 6], GHCJS [18] and Curry [11]. Because browsers that support JavaScript usually also expose their HTML DOM through a JavaScript API, this solution has as an additional advantage that the HTML DOM is easily accessible from within the functional code.

The solution of compiling Clean, Haskell or Curry to JavaScript also has disadvantages. One has to rely on JavaScript garbage collection, which is not optimized for the way non-strict functional programming languages use memory. Furthermore, programs in functional languages often are deeply recursive, which can lead to overflow problems in JavaScript. The resulting code is also mostly much slower than its server-side counter part (varying from 10 to 60 times slower).

In this paper we demonstrate an entirely different and more portable approach. Instead of compiling to JavaScript directly we introduce a portable Virtual Machine JMVM with a corresponding bytecode format that is capable of executing the bytecode that is generated by a dedicated compiler for the non-strict intermediate functional language Sapl. Sapl has been used before as an intermediate language for the translation of Haskell and Clean to JavaScript [2, 4].

For this Virtual Machine several implementations have been made, including versions in C(++) and JavaScript. The different versions use the same bytecode instructions generated by the aforementioned compiler. The JavaScript versions of the VM includes both a plain JavaScript version and an asm.js version. For asm.js we made a hand-coded and an Emscripten version [25]. The performance of the asm.js VM turns out to be competitive. It is faster than all previous interpreters we made for Sapl and is on par with Haste. The C(++) version is, by our knowledge, the fastest existing interpreter for non-strict functional languages.

Summarizing, we obtained the following results:

- We realized a simple stack-based Virtual Machine JMVM with implementations in C(++) and JavaScript (including asm.js);
- The JMVM bytecode formalism includes only a few super instructions that are especially needed for the efficient execution of non-strict functional languages. All other instructions are of a generic nature as can be found in other Virtual Machines like JVM, etc;
- A dedicated compiler for the intermediate language Sapl to JMVM bytecode format has been realized. Due to the addition of the extra super instructions code generation is simple and completely straightforward;
- Executing compiled code on the C(++) implementation of the VM is faster (more than 3 times) than for the original Sapl interpreter [13]. The C(++) implementation has a performance that is only 3 times slower than Clean. The asm.js implementation is averagely less than 8 times slower than Clean;
- Executing code on the asm.js based VM is faster than for all previous versions that used JavaScript generation and does not suffer from stack overflow problems that occurred in these approaches;
- The JavaScript and asm.js versions of the VM are small (less than 1000 lines of code, or 25kB) and can be loaded quickly;
- The size of the byte code of compiled programs is a tenth of that of programs that are compiled to JavaScript;
- The VM allows for dynamic linking of additional code and can (therefore) also be used for the realization of an interpreter (with a read-evaluate-print-loop) and for scripting languages.

The layout of this paper is as follows. In Section 2 we give a motivation for this work. Section 3 describes the Virtual Machine. In Section 4 the Sapl programming language is described. This is the source language for our compiler which is described in Section 5. In Section 6 we present benchmark tests for the C(++), JavaScript and asm.js versions of the VM. We compare our approach with work of others in Section 7. Finally, we end with conclusions and a look ahead in Section 8.

## 2 ALTERNATIVE EXECUTION PLATFORMS FOR CLEAN AND HASKELL

Clean and Haskell both have native code compilers for a variety of computer platforms (Intel, ARM) and operating systems (Windows 10, Linux, MacOS, Android, IOS) These compilers allow these languages to run at maximum speed on these platforms.

But native code cannot be run in web-browsers because these browsers are still lacking a universal assembly-like formalism to execute code. The only platform available is the scripting language JavaScript. Previously, we already developed a Clean to JavaScript compiler for the execution of Clean code at the client-side [2]. Its most important usage is for dealing with interactive objects like handling SVG-objects and so-called EditLets [3]. Clean uses a dedicated back-end in its compiler to generate JavaScript code.

Similar approaches were used in Haste [5] and GHCJS [18] to compile Haskell to JavaScript. Both generate JavaScript from intermediate formats like STG or Haskell core.

Unfortunately, the generated JavaScript code runs 10-60 times slower than native Clean code, with a great variety in performance and suffer from stack overflow problems. Although, this speed is good enough for many cases, we run into troubles for complex SVG generation.

## 3 THE JMVM VIRTUAL MACHINE

Virtual machines have become popular as execution platform for programming languages. Especially, the Java Virtual Machine [17] is a successful example of such a platform. Besides Java, many other programming languages are implemented on top of the JVM, including the (non-lazy) functional language Scala [19]. A Virtual Machine forms an extra abstraction layer on top of specific hardware platforms. They mostly have smaller and easier to use instruction sets that are independent of the platform the VM has to run on. Virtual machine implementations of programming languages used to be slower than direct implementation, but with the rise of embedded JIT compilers they have become of competitive performance. Ertl et al. [7] give an extensive description of how efficient VM bytecode interpreters can be made.

### 3.1 Motivation for a dedicated Virtual Machine for non-strict languages

Of course, existing VM's like the JVM can be used to implement modern non-strict functional languages. For example, Frege [24] is an example of a Haskell like language that runs on top of the JVM. But, although it is still possible to run JVM based applications within all major web-browsers using the JVM plug-in, this has become less popular in recent years, due to security problems. Another issue is that not all portable platforms, like tablets and smart-phones, support the JVM plug-in.

Because we do not want to rely on plug-ins to run the client part of a Clean application, we have to rely on the single execution platform that is available in all web-browsers: JavaScript. Together with the performance issues, mentioned earlier, this motivated our choice to develop a dedicated portable Virtual Machine for the execution of non-strict functional languages that can run at the client-side using only JavaScript.

## 3.2 Definition of the JMVM bytecode

The bytecode interpreter is mainly inspired by the JVM, but also partly by the G- and STG-machines [15, 20]. It defines a stack based virtual machine that is equipped with only a few dedicated instructions aimed at the efficient interpretation of (non-strict) functional code.

*3.2.1 Stacks.* The VM has 2 stacks: a value and a control stack. The control stack is used for return addresses and stack administration (base pointer). All computations are done on the value stack. Basic values (int, float, bool and char) and pointers (to heap blocks) are distinguished by using a bit based tagging scheme. All stack values are shifted one bit to the left. Basic values have a right most bit 0 and pointers a right-most bit 1.

*3.2.2 Memory management and Garbage Collection.* Memory is divided into: the (control) stack; the heap and the constant pool. The constant pool contains strings that appear in the source code. The heap contains memory blocks that are created by the create instructions. Each heap block has a description field that can be inspected by certain instructions. Garbage collection is automatic. In the current implementation a copying, or two space, garbage space collector is used. If memory is exhausted, all heap blocks that are reachable from the stack (using the bit tag) and the heap blocks that are recursively reachable from these blocks are copied to a new semispace of the heap. After this the old semispace becomes available for the next GC round. In a copying garbage collector memory is compacted after every GC. It also offers the possibility to implement extensible records and arrays in a relatively easy way (not used at this moment).

## 3.3 Detailed discussion of the bytecode instructions

Most byte code instructions are similar to those in stack-based virtual machines like JVM. We distinguish: stack handling instructions, arithmetic instructions, (conditional) branching instructions including case tables, (tail) call instructions, memory creation and manipulation instructions and string handling instructions. We discuss them all briefly. The instructions that were made dedicated for the efficient execution of non-strict FP languages are dealt with in detail separately.

Table 1 gives an operational description of the JMVM instructions that are used in the examples of this paper. The state of the virtual machine can be described by the following state variables:

- st the stack, sp the stack pointer (points to top of stack, stack grows to higher numbers);
- cst the control stack, used for storing return addresses and the base pointer, csp the control stack pointer;
- heap the heap, hp the heap pointer (points to the next free heap segment);
- prog the program store, pc the program counter (points to the current instruction);
- bp the base pointer (points to the start of the stack frame for the current function).

In the instruction table the following abbreviations are used:

- op stands for one of the arithmetic instructions add, sub, etc;
- ifop stands for one of the if-instructions iflt, ifeq, etc;
- top is equivalent to st[sp];
- new(n) creates a heap block of size n and type Array and returns a pointer to this block;
- create(n,t,c,s) creates a heap block of size n, copies n values from the stack starting at s to this block and tags the block with size n, type t, and constructor type c and returns a pointer to it. If p is a pointer to a heap block then: size(p) returns its size, type(p) its type and cons(p) its constructor type;
- copy(h,s,n) copies a heap block of size n starting at h to the stack starting at s;
- slice(a,b) slices a segment from the stack starting at a and ending at b (remove this part and shift the part above it);
- pushcs(addr,bp) pushes addr and bp onto the control stack;
- (pc,bp) = popcs() pops values for pc and bp from the control stack;
- print(v,t) prints value v using formatting t (int, float, string, etc);
- ns[a] in jmpt and jmptable: take the a'th element of ns.

*3.3.1 Stack operations.* Before a function is called, its arguments must be pushed onto the stack. In the function code arguments can be loaded by load n (load the n-th element relatively to the base pointer on top of the stack). An element at the top of the stack can be stored at location n with store n (overwriting the existing value). The VM further supports the following stack operations: push constant, pop, swap, dup, updsp.

*3.3.2 Arithmetic operations.* Arithmetic operations operate on elements on the stack and put their result back on the top of the stack. We currently support the standard integer operations: add, sub, mult, div, mod, inc, dec and the float operations: fadd, fsub, fmult, fdiv, sin, cos, tan, asin, acos, atan, log, log10, exp, sqrt, floor, pow.

*3.3.3 (Conditional) Branching.* The VM has standard instructions for (conditional) branching: ifeq, ifneq, ifcmp, iflt, ifle, jmp, with their expected meaning.

jmptable n 1 L2 .. Ln: can be used to implement switch-statements, n is the number of cases, L1 L2 .. Ln are the labels to jump to. If k is on top of the stack, the k'th label is selected.

jmpt is a special instruction for handing case expressions in a functional language and is described in detail in Sec. 3.4.3.

*3.3.4 Function handling.* The VM has a call label instruction, which pushes the return address and the base pointer onto the control stack and moves execution to label. The instructions return nrargs and return_const nrargs val both return control to the callee after slicing nrargs elements from the stack and putting the result of the call back on top of the stack.

pushfunc nrargs label pushes a function value onto the stack. A call for a function value on top of the stack can be issued by using the apply instruction (the arguments of the function should also be on the stack, like in call).

**Table 1: JMVM instructions**

| Instruction | arg1 | arg2 | arg3 | Semantics | PC | SP |
|---|---|---|---|---|---|---|
| **Stack** | | | | | | |
| push | v | | | st[sp+1] = v | pc+2 | sp+1 |
| pushfunc | na | addr | | st[sp+1] = addr << 32 + na | pc+3 | sp+1 |
| swap | | | | top = st[sp-1] & st[sp-1] = top | pc+1 | sp |
| updsp | n | | | | pc+2 | sp+n |
| dup | | | | st[sp+1] = top | pc+1 | sp+1 |
| load | n | | | st[sp+1] = st[bp-n] | pc+2 | sp+1 |
| store | n | | | st[bp-n] = top | pc+2 | sp-1 |
| inc | | | | top = top + 1 | pc+1 | sp |
| dec | | | | top = top - 1 | pc+1 | sp |
| loadadd | n | v | | st[sp+1] = st[bp-n] + v | pc+3 | sp+1 |
| op | | | | st[sp-1] = st[sp-1] op top | pc+1 | sp-1 |
| **Branching** | | | | | | |
| ifop | addr | | | if (top op 0) pc = addr else pc = pc + 2 | - | sp-1 |
| ifcmp | addr | | | if (top == st[sp-1]) pc = addr else pc = pc + 2 | - | sp-2 |
| ifltlv | n | v | addr | if (st[bp-n] < v) pc = addr else pc = pc + 4 | - | sp |
| jmp | addr | | | | addr | sp |
| call | addr | | | pushcs(pc+2,bp) bp = sp | addr | sp |
| tailcall | na | addr | | slice(bp-na,bp) bp = sp-na | addr | sp-na |
| jmptable | n | ns | | pc = ns[top] | - | sp-1 |
| jmpt | n | ns | | pc = ns[cons(top)] copy(top,sp,size(top)) | - | sp+size(top)-1 |
| return | n | | | st[bp-n+1] = top (pc,bp) = popcs() | - | sp-n+1 |
| return_const | n | v | | st[bp-n+1] = v (pc,bp) = popcs() | - | sp-n+1 |
| apply | | | | pushcs(pc+1,bp) bp = sp - 1 | top>>32 | sp-1 |
| **Heap** | | | | | | |
| create | n | t | | st[sp-n+1] = create(n,t,0,sp) | pc+3 | sp-n+1 |
| ccreate | n | t | c | st[sp-n+1] = create(n,t,c,sp) | pc+4 | sp-n+1 |
| get | n | | | top = heap[top+ n] | pc+2 | sp |
| getn | n | | | st[sp-1] = heap[top + st[sp-1]] | pc+1 | sp-1 |
| update | | | | heap[st[sp-1] + st[sp-2] ] = top | pc+1 | sp-3 |
| getall | | | | copy(top,sp,size(top)) | pc+1 | sp+size(top)-1 |
| newarray | | | | st[sp] = new(top) | pc+1 | sp |
| **Aux** | | | | | | |
| stop | | | | | stop | stop |
| print | t | | | print(top,t) | pc+2 | sp-1 |

tailcall nrargs label can be used for tail calls (do not push a return address onto the control stack and remove current stack-frame by slicing).

*3.3.5 Heap handling.* With create nrelems type a memory block is created on the heap. The block is filled with the top nrelems elements from the stack. create returns a pointer to the memory block. The type field is used to tag the heap block (examples can be found in Sec. 5).

ccreate nrelems type ctag is meant for constructor creation and has an argument ctag indicating the index of the constructor.

An empty array can be created with newarray size.

Individual heap elements can be accessed by using either: get index, with the pointer to the heap block on top of the stack or with getn, with the pointer and index both on the stack. Values in a heap block can be updated using: update, with the pointer, index and value on the stack.

The getall instruction is the inverse of the create instruction. It puts back the elements of a heap block on the stack in the same order as during creation.

*3.3.6 Types of heap blocks.* The following predefined types for tagging heap blocks are currently used in the VM: Record, Thunk, Closure, Constr, Forward, String and Boxed. Record is also used for arrays. A thunk is a block representing a saturated function call (there are exactly as many arguments as the function needs). A closure represents an under- or over-saturated function call. It is the responsibility of the compiler writer or programmer to ensure that a thunk or closure is made by first pushing arguments on the stack followed by a function value using pushfunc. Forward is used during garbage collection. Boxed is used to indicate a boxed value (see Sec. 3.4.1).

*3.3.7 String handling.* The VM has a number of instructions for the efficient handling of strings. strcreate, strappend, strat, strupdate, strslice, strcpy, strlen, strcmp. Because they are not used in the examples we do not describe them here in detail.

*3.3.8 Other instructions.* stop stops execution. print type prints the top of the stack formatted as type (int, float, char, bool, string). For printing ADT's, Records, etc, explicit code must be written.

*3.3.9 Super Instructions.* An important way to optimize virtual machines is the use of so-called super instructions. Super instructions combine the code of a number of normal instructions that are often used in sequence. An example of a super instruction is the loadadd index constant instruction, which loads a value from the stack and immediately adds a constant to it. Another example is the ifltlv index constant label instruction that combines a load and a branch. The current implementation only makes modest use of super instructions, so there is still room for performance improvements.

## 3.4 Dedicated FP-instructions

With the instruction set mentioned so far, it is possible to implement a simple imperative programming language. For the implementation of a non-strict functional programming language it is necessary to handle functions as data and to deal with laziness by means of the creation of thunks and closures and the (delayed) evaluation of them. This can be realized using the create, getall, apply and update instructions, but life for a compiler writer can be made a lot easier by adding special super instructions for this.

*3.4.1 Dealing with laziness and thunks.* A thunk is a heap block representing a delayed function call. Thunks can be created by using the create instruction (assuming that a function value and its arguments reside on the stack). A thunk can be evaluated by using getall and apply. Because thunks can be passed as arguments and thus can be shared, the result of the evaluation of a thunk should be written back to the thunk block by making an update (at index 0) to obtain a so-called boxed value. The type tag of the heap block should be adapted to distinguish boxed values from thunks. When trying to evaluate a thunk it should therefore always be checked whether it is already a boxed value. In that case the boxed value must be returned. Because thunks (or boxed values) may return another thunk, repeated evaluation of thunks may be necessary until a value in head-normal-from is reached.

Beside thunks that contain exactly all necessary arguments, there are also closures: under- (curried) or over-saturated calls. A curried closure can be applied to other elements available on the stack. In the case of evaluation of closures the result should not be written back as a boxed value.

*3.4.2 The eval instruction.* For efficiency reasons a dedicated eval super-instruction is added to the virtual machine. This instruction offers an efficient implementation for the evaluation of thunks and closures to realize non-strict behaviour. The eval (Fig. 1 shows its pseudo code) instruction combines all functionality mentioned above in one super instruction. Because eval must be applied repeatedly until a normal form has been reached, the address of the current eval instruction must be pushed onto the control stack, for returning back to it, as long as there is a thunk or closure on top of the stack. When evaluating a thunk also a return address for a boxvalue must be pushed onto the control stack. The boxvalue instruction takes care of updating the thunk with the result of the evaluation and updating its type tag. It immediately returns control back to the corresponding eval by popping its return address from the control stack. boxvalue is not intended to be used directly in code!

In case of a curried closure, it must be checked whether there are indeed enough arguments on the stack. If this is the case, the call is made. Otherwise a new curried closure, containing the available elements, is returned.

eval can always be applied to any value on the stack with minimal overhead! In case the top of the stack is not a thunk or closure eval only advances the pc to the next instruction. The use of the eval instruction simplifies the code generation process significantly. Evaluation of thunks, the application of function variables to arguments and the handling of closures can now all be dealt with in the same way using this instruction. Experiments have shown that the extra overhead in using eval instead of apply is hardly noticeable.

*3.4.3 The jmpt instruction.* Another super instruction, especially added for the efficient dealing with pattern matching, is jmpt n L1 L2 .. Ln. This is a variant of the jmptable instruction. jmpt assumes a pointer to a constructor on top of the stack. The jump is determined by the ctag field of the constructor and the arguments of the constructor are pushed onto the stack. So jmpt combines getctag, jmptable and getall in one instruction.

Examples of the use of eval and jmpt can be found in Sec. 5.3.

## 3.5 The implementation of JMVM

JMVM is implemented using standard techniques for the construction of efficient virtual machines as described in [7]. For the C(++) version a threaded implementation using computed goto's is made.

```
void eval() {
  stackvalue current = tos, tmp;
    for(;;)
      switch (type(current)) {
        case value: case record: case array: case string:
          return current;
        case boxed:
          current = current[0]; // unbox
          break;
        case thunk:
          tmp = apply(getall(current));
          current[0] = tmp;     // box result
          current = tmp;
          break;
        case closure:
          if (enough_args_on_stack)
            current = apply(getall(current));
          else
            return new_closure_expr;
          break;
      }
}
```

**Figure 1: Pseudo code for the eval instruction**

This means that the main execution function consists of one big code block with labelled sub-blocks representing the different instructions. Jumping between these blocks is done by goto statements. This version of the interpreter is about 30-40% faster than one using traditional C-switch statements. The stack and the heap are pre-allocated arrays of bytes using 64 bits words (but the VM can also be compiled as a 32-bit architecture). Memory management is done using a copying garbage collector.

In the JavaScript version we cannot use computed goto's and have to use a traditional switch statement. The stack and heap are modelled as big pre-allocated byte arrays (typed arrays in JavaScript). Also here, memory management is explicit using a copying garbage collection, so no use is made of the JavaScript garbage collector. The JavaScript version currently only supports 32-bit values.

For the C(++) VM implementation we can choose between a 32- or 64-bit architecture. Using a 64-bit architecture, integers are actually 63 bits (because of the pointer bit). Because float values lose their sign when shifting back and forth one bit, we use 32-bit floating point numbers embedded in a 64-bit word.

In a 32-bit implementation we need another way to represent floating point numbers There are several possible solutions for this, the most straightforward one is using indirections and store floating point numbers as boxed values on the heap.

*3.5.1 An asm.js version of JMVM.* The JavaScript implementation shows a great variety in performance over the different web-browsers (see Sec. 6). We therefore decided to also make a dedicated asm.js [12] implementation, in the hope that this yields better results. Asm.js consists of a strict subset of the JavaScript language and omits many of JavaScript's dynamic features. Programs are statically typed and types are restricted to numerical values. Programs in asm.js must use explicit memory management

```
::list = Nil | Cons !x xs

nfib !n = if (n < 2) 1 (nfib (n-1) + nfib (n-2) + 1)

start  = fac 6
facl n = if (n == 0) 1 (n * facl (n-1))
fac !n = if (n == 0) 1 (n * fac (n-1))

twice f x = f (f x)
twicex = twice twice twice twice inc 0
inc !n = n + 1

sieve xs
= case xs (Nil -> Nil)
          (Cons a as -> Cons a (sieve (filter (notdiv a) as)))
primes     = sieve (from 2)
notdiv n m = m % n != 0

filter p xs = case xs (Nil -> Nil)
                      (Cons a as -> if (p a)
                                       (Cons a (filter p as))
                                       (filter p as))

hamming = let h  = Cons 1 (merge (merge as bs) cs),
              as = map (mult 2) h,
              bs = map (mult 3) h,
              cs = map (mult 5) h  in h
mult !a !b = a * b
```

**Figure 2: Example Sapl programs**

using one big binary heap. Due to this, an optimized JavaScript engine can compile and optimize asm.js code ahead of time (before executing the code). Because asm.js is a strict subset of JavaScript it is also executable by JavaScript engines not supporting asm.js optimizations.

Asm.js delivers near native performance (1.5 - 2 times slower) for typical C(++) programs that are compiled to asm.js by using the Emscripten compiler. Emscripten [25] is a dedicated compiler for translating C(++) (actually it translates LLVM) to asm.js. Using Emscripten to compile the C(++) version of the VM results in a slowdown of more than a factor of 3. The resulting code size is 8 times bigger than the source code. We therefore decided to develop a hand-coded asm.js version of the VM also. The performance of this version is better than that of the JavaScript and Emscripten versions and also varies a lot less for the different web-browsers (see Sec. 6).

## 4 THE SAPL PROGRAMMING LANGUAGE

Sapl is a simple intermediate, but still high-level, non-strict functional programming language. Sapl emerged in combination with a simple efficient interpreter written in C(++) and based on straightforward graph-reduction techniques [13]. Sapl is almost a subset of Clean or Haskell (only minor syntactic differences). We use Sapl as an intermediate language for translating Clean (the Clean compiler has a Sapl back-end) either directly to JavaScript [2] and in this paper also to bytecode for the Virtual Machine.

Recently, direct support for strings and string operations, records and arrays with select and update operations and strictness annotations, were added to Sapl.

Although, Sapl is used here as an intermediate core language, it is at a higher level than Haskell Core, STG and Core Clean. Its syntax supports infix arithmetic expressions and it is very straightforward to program in Sapl. The main difference with Clean and Haskell at the syntax level is the absence of: list-comprehensions, pattern matching, overloading, $\lambda$-expressions and local function definitions. Sapl programs are not type checked, but Sapl programs generated by the Clean compiler are correctly Hindley-Milner typed.

Figure 2 shows examples to illustrate the syntax of Sapl. We briefly describe the main constructs of Sapl (records and arrays are not treated, because they are dealt with similarly as instances of ADT's). A simplified syntax for Sapl is given by:

| | | | |
|---|---|---|---|
| $D$ | ::= | $:: x = C\ \vec{x} \mid \ldots \mid C\ \vec{x}$ | (Algebraic Type Definition) |
| $f$ | ::= | $x\ \vec{x} = e$ | (Function Definition) |
| $e$ | ::= | $x\ \vec{e}$ | (Application) |
| | \| | **case** $e\ \vec{e}$ | (Case Expression) |
| | \| | **if** $e\ e\ e$ | (If Expression) |
| | \| | **let** $\vec{b}$ **in** $e$ | (Let Expression) |
| | \| | $C\ \vec{x} \to e$ | (Case) |
| | \| | $x$ | (Variable) |
| | \| | $l$ | (Literal) |
| $b$ | ::= | $x = e$ | (Let Binding) |

**Algebraic Type Definitions** Algebraic type definitions provide the names of the constructors, their arity and their position with respect to the other constructors (used by the case construct). Constructors should always be used fully saturated.
**Function Definitions** A function definition consist of the function name, the arguments and a function body.
**Applications** Sapl does not require applications to be saturated (they may be over and under saturated). Arguments of applications are allowed to be other applications.
**Case Expressions** `Case` expressions are intended to perform pattern matching. The `case` keyword is used to make a case analysis on the data type of its first argument. To accomplish this, the first argument must be reduced to *head-normal-form*. The remaining arguments handle the different constructor cases (all possible cases must be handled separately and in the same order as in the type definition). Each case consists of the constructor name applied to argument variables, followed by the case body.
**If Expressions** An `if` expression can be considered as a normal function with strict semantics in its first argument (which must have a boolean result). It returns its second (true-case) or third argument (false case).
**Let Expressions** Only constant (non-function) `let` expressions are allowed. Let definitions may be mutually recursive (for creating cyclic expressions).
Variables can be annotated as being strict with (!). This can happen at:

- the arguments of a function definition;
- the left-hand side of a let definition;
- the arguments of constructors in ADT's.

## 5 A SAPL TO JMVM COMPILER

Due to the choice of the instruction set (especially eval and jmpt) the compiler can be kept small and elegant. The basic idea is to implement a strict compiler and only deviate from this when necessary. Therefore, strictness annotations are important in this process.

We illustrate the compilation process by using 3 examples: the strict and non-strict variants of fac; the sieve function; the twice and twicex functions (see Fig. 2). Together they cover all important aspects that have to be dealt with in the compiler.

### 5.1 Transformation to explicit form

The translation of Sapl to JMVM code proceeds in a number of steps. We start with a number of source to source transformations that transform Sapl code into a form that can be translated to JMVM straightforwardly.

Before doing this, we first give a taxonomy of the different kinds of applications that can occur in Sapl programs. An application is either one of the following:

- an application of a known function to exactly the right number of arguments (saturated call);
- an application of a known function to too few arguments (under-saturated call);
- an application of a known function to too many arguments (over-saturated call);
- an application of an unknown function (variable or unknown closure) to a number of arguments.

Again, we do not treat records and arrays, because they are dealt with similarly as instances of ADT's.

*5.1.1 Steps in the transformation process.* In the transformation process we always consider one function at a time, so the context of the transformation process is always a single function.

*5.1.2 Step 1.* In this step Boolean expressions and if-statements are transformed into a form that matches the conditional instructions in JMVM. In this step conjuncts and disjuncts are replaced by nested if-statements.

This step is standard for all compilers (even for non-functional languages). Therefore, we omit the details of this step.

*5.1.3 Step 2.* In this step sub-expressions that appear in a non-strict context are identified. Our goal was to proceed as much as if we were dealing with a strict language. We therefore have to identify the places where we have to deviate. The following sub-expressions are tagged with the keyword lazy (they occur in a non-strict context):

- non-strict arguments of known function applications;
- bodies of non-strict let-expressions;
- arguments of under-saturated applications;
- arguments of unknown applications;
- over-saturated arguments of over-saturated applications;
- non-strict arguments of constructors.

Fig. 3 shows the result of tagging lazy for the examples.

*5.1.4 Step 3.* In this step we $\lambda$-lift lazy tagged sub-expressions to new functions, if necessary. Lifting is **not** necessary for:

- literals;

```
facl n = if (n == 0) 1 (n * facl (lazy (n-1)))
fac !n = if (n == 0) 1 (n * fac (n-1))

twice f x = f (lazy (f (lazy x)))
twicex = twice (lazy twice) (lazy twice) (lazy twice) (lazy inc) 0
inc !n = n + 1

sieve xs
= case xs
   (Nil -> Nil)
   (Cons a as ->
      Cons a
       (lazy (sieve (lazy (filter (lazy (notdiv (lazy a)))
                                  (lazy as))))))
```

**Figure 3: Sapl examples with lazy tagging**

```
facl n   = if (n == 0) 1 (n * facl (lazy (facl_1 n)))
facl_1 n = n - 1
```

**Figure 4: Lifted Factorial in Sapl**

- variables;
- applications (but lifting may be necessary for arguments in the application);
- constructors with only non-strict arguments (again arguments may be lifted);

In all other cases, expressions must be lifted.

During lifting the expression is replaced by a call to a function applied to the variables that occur in the expression. In our examples only in the non-strict version of factorial, the lazy tagged sub-expression (n-1) needs to be lifted to a new function. Fig. 4 shows the result.

Lifting is a standard, and often used technique, that is also used to replace local functions by global functions. Note, that in this transformation lazy tagged sub-expressions are always transformed into non-lazy top-level expressions.

Due to lifting, lazy tagged if- and case-expressions, lazy tagged primitive (numeric and string) operations and lazy tagged constructors with strict-arguments are always moved to new functions and can therefore never be tagged lazy any more! Only function applications, constructors with only non-strict arguments and variables can be tagged lazy after lifting.

*5.1.5 Step 4.* In this step sub-expressions are possibly (re-)tagged with either: thunk, closure, eval to indicate what should happen with them at runtime.

- lazy tagged saturated applications are re-tagged thunk;
- lazy tagged under-, over-saturated and unknown applications are re-tagged closure;
- untagged under-saturated applications are also tagged closure;
- untagged over-saturated and unknown applications are tagged eval;
- untagged non-strict variables are tagged eval;

```
facl n   = if (eval(n) == 0)
              1
              (eval(n) * fac (thunk (facl_1 n)))
facl_1 n = eval(n) - 1

fac !n = if (n == 0) 1 (n * fac (n-1))

twice f x = eval (f (closure (f x)))
twicex = eval (twice (closure twice) (closure twice)
                     (closure twice) (closure inc) 0)
inc !n = n + 1

sieve xs
= case (eval xs)
   (Nil -> Nil)
   (Cons a as ->
      Cons a
        (thunk (sieve (thunk (filter (closure (notdiv a)) as)))))
```

**Figure 5: Fully annotated Sapl examples**

All other lazy tagged expressions lose their tag (constructors and lazy tagged variables). Fig. 5 shows the results of the re-tagging. As a result of the transformation process we have obtained a fully explicit program. The places where thunks or closures have to be created are made explicit. Due to $\lambda$-lifting only applications can be tagged with thunk or closure, so there is no need for the storage of more complex graphs. Also the places where thunks and closures have to be evaluated are made explicit. The functions are now ready for translation to JMVM.

**Note**: Numeric expressions that occur in a lazy context are always lifted to new functions. For numeric expressions that only contain literals and strict variables, it is probably cheaper to make them strict, instead of lifting them. But this may cause troubles in case of zero division or integer overflow.

## 5.2 Compilation to JMVM

Generation of JMVM can now be realized by a straightforward tree walk over the parsed syntax tree. Functions are translated one-by-one. A function that has strict arguments must have two entry points:

- for (direct) calls for which the strict arguments are put evaluated on the stack;
- for calls through evaluation of thunks or closures, for which the strict arguments still have to be evaluated.

This 'lazy' entrance is put before the 'strict' entrance (see the factorial example in Sec. 5.3.1).

In Table 2: `a,b,c` stand for expressions, `as` for an application; `f as` for the application of `f` to `as`; `cs` for a sequence of variables or expressions (depending on the context); `v` for a variable; `ind(v)` for the stack offset of variable `v`; `f` for a function. $\mathtt{C}_t$ for constructor `C` with tag `t`; `n` for a literal; `framesize` for the size of the current stack frame; `|as|` for the size of `as`; $E$(`a`)`| a <- as` for $E(\mathtt{a}_1)$ + … + $E(\mathtt{a}_n)$; `code as` for $E$(`a`)`| a <- reverse as`; `if` for an if-instruction that matches the condition expression; `prim` for a primitive operation; `ls` for a sequence of let expressions; `lb:a` for

**Table 2: Sapl to JMVM compilation rules**

| | | |
|---|---|---|
| $T$(`if a b c`) | = | $E$(`a`) + `if lb` + $T$(`c`) + `lb:`$T$(`b`) |
| $T$(`f as`) | = | `code as + tailcall framesize f` |
| $T$(`case a cs`) | = | $E$(`a`) + `jmpt n ls` + $S$(`c`) `\|c <- cs` |
| $T$(`b`) | = | $E$(`b`) + `return nrargs` |
| $S$(`C cs -> a`) | = | `lb:`$T$(`a`) |
| $E$(`n`) | = | `push n` |
| $E$(`v`) | = | `load ind(v)` |
| $E$(`f`) | = | `pushfunc f` |
| $E$(`closure as`) | = | `code as + create \|as\| closure` |
| $E$(`thunk as`) | = | `code as + create \|as\| thunk` |
| $E$(C$_t$ `as`) | = | `code as + ccreate \|as\| constr t` |
| $E$(`f as`) | = | `code as + call f` |
| $E$(`eval (f as)`) | = | `code as + call f + eval` |
| $E$(`eval as`) | = | `code as + eval` |
| $E$(`eval c`) | = | $E$(`c`) + `eval` |
| $E$(`if a b c`) | = | $E$(`a`) + `if lb` + $E$(`c`) + `lb:`$E$(`b`) |
| $E$(`prim as`) | = | `code as + prim` |
| $E$(`let ls in a`) | = | `upsp \|ls\|` + $L$(`l`)`\|l<-ls` + $E$(`a`) |
| $L$(`v = a`) | = | $E$(`a`) + `update ind(v)` |

a code-block labelled with lb.
The compilation rules are schematically given in Table 2. We start with $T$(b), with b the body of the function, $T$ is used for compiling top-level expressions and $E$ for other expressions). If a top-level expression is a function call it will be turned into a tail call.

Let-expression are assumed to be sorted on dependency. In case of let-expressions that are cyclic dependent on each other, a cycle analyser in the compiler will break the cycle by substituting a dummy value for a reference to one of the let-variables. During execution of the code the dummy value is replaced (using the update instruction) by the reference to the correct let after all let-bodies are instantiated.

## 5.3 Code examples

Here we show the results of applying the compilation rules to the examples. All code is taken from the actual compiled programs. Function names are added in comment at the beginning of function code blocks and also at the calls themselves.

*5.3.1 Strict Factorial.* (Fig. 6) Because fac has a strict argument, there is a non-strict entry-point added before the actual code (it is never used). fac can be executed entirely on the stack. There is no creation of thunks or closures. The code is equivalent to the code a compiler should have generated for a strict language. The code also includes the start function containing a call to fac 6.

*5.3.2 Non-strict Factorial.* (Fig. 7) In this case thunks are created and evaluated for the argument of facl.

*5.3.3* `notdiv` *and* `sieve`. (Fig. 8) In this example laziness is really needed to deal with an infinite list of numbers.

*5.3.4* `twice` *and* `twicex`. (Fig. 9) This is the most complex example because of the over-saturated call for twice in twicex. This is also the only example that really needs the repetitive implementation of eval. In this example eval is both used for evaluating thunks and for applying closures to arguments that can be found on the stack.

```
            call L0                 || call start
            print int
            stop
0           push 6                  || start
            tailcall 0 L2           || call fac
1           load 0                  || fac_lazy
            eval
            store 0
2           load 0                  || fac
            ifneq L3
            return_const 1 1
3           load 0
            dup
            push 1
            sub
            call L2                 || call fac
            mult
            return 1
```

**Figure 6: Strict Factorial**

```
            call L0                 || call start
            print int
            stop
0           push 6                  || start
            tailcall 0 L1           || call facl
1           load 0                  || facl
            eval
            ifneq L2
            return_const 1 1
2           load 0
            eval
            load 0
            pushfunc 1 L3           || push facl_1
            create 2 thunk
            call L1                 || call facl
            mult
            return 1
3           load 0                  || facl_1
            eval
            push 1
            sub
            return 1
```

**Figure 7: Non-strict Factorial**

## 6 BENCHMARKS

In this section we present the results of several benchmark tests for the VM implementation of Sapl. The benchmark programs we use for the comparison are the same as the benchmarks we used for comparing the original Sapl interpreter in C++ with other interpreters and compilers in [13]. In that comparison it turned out that Sapl is at least twice as fast (and often even faster) as

```
11          load 1              || notdiv
            eval
            load 0
            eval
            mod
            ifeq L12
            return_const 2 1
12          return_const 2 0
16          load 0              || sieve
            eval
            jmpt 2 L17 L18
17          ccreate 0 constr 0  || create Nil
            return 1
18          load 1
            load 0
            pushfunc 2 L11      || push notdiv
            create 2 closure
            pushfunc 2 L9       || push filter
            create 3 thunk
            pushfunc 1 L16      || push sieve
            create 2 thunk
            load 0
            ccreate 2 constr 1  || create Cons
            return 3
```

**Figure 8: Sieve and notdiv**

```
6           load 1              || twice
            load 0
            create 2 closure
            load 0
            eval
            return 2
7           load 0              || twicex
            pushfunc 1 L4       || inc_lazy
            create 1 closure
            pushfunc 2 L6       || twice
            create 1 closure
            pushfunc 2 L6       || twice
            create 1 closure
            pushfunc 2 L6       || twice
            create 1 closure
            call L6             || twice
            eval
            return 1
```

**Figure 9: Twice**

other interpreters like Helium, Amanda, GHCi and Hugs. For the implementation of the native version of the VM we used the clang compiler (gcc gives similar results) on a macbook pro 2013 (Sierra, 2.4 GHz Intel Haswell Core i5). For both the native and client-side version of JMVM the heap size was 50MB (per semispace) and the stack size 4MB.

We briefly repeat the description of the benchmark programs here:

(1) **Nfib** The (naive) Fibonacci function, calculating `nfib 38` (see Fig. 2).
(2) **Primes** The prime number sieve program, calculating the 5000th prime number (see Fig. 2).
(3) **Queens** Number of placements of 11 Queens on a 11 * 11 chess board.
(4) **Twice** The famous higher order function twice (see Fig. 2), repeated 400 times.
(5) **Knights** Finding a Knights tour on a 5 * 5 chess board.
(6) **Match** Nested pattern matching (5 levels deep) repeated 3000000 times.
(7) **Sprimes** Symbolic prime number sieve using Peano numbers, calculating the 280th prime number.
(8) **Eval** A small Sapl interpreter. As an example we coded the prime number sieve for this interpreter and calculated the 100th prime number.
(9) **Hamming** The generation of the list of Hamming numbers (a cyclic definition, see Fig. 2) and taking the 1000th Hamming number, repeated 4000 times.
(10) **Parser** A parser for Prolog programs based on Parser Combinators parsing a 35000 lines Prolog program.
(11) **Prolog** A small Prolog interpreter based on unification only (no arithmetic operations), calculating ancestors in a four generation family tree.
(12) **Sort** Quick Sort (10000 elements), Insertion Sort (6000 elements), Merge Sort (100000 elements, because merge sort is much faster, we use a larger example). For sorting a list of size $n$ a source list is used consisting of numbers 1 to $n$. The elements that are 0 modulo 10 are put before those that are 1 modulo 10, etc.

The benchmarks cover a wide range of aspects of functional programming (lists, laziness, deep recursion, higher order functions, cyclic definitions, pattern matching, heavy calculations, heavy memory usage). All times are machine measured. On the client-side the tests were repeatedly executed until a stable execution time was obtained. This to give the JIT compiler time to do its job.

The complete code of all benchmarks and their translation to JMVM can be found at [14].

*6.0.1 Benchmark Tests.* We ran the tests using the following compilers or interpreters:

- Sapl: the original Sapl interpreter in C++;
- Clean: the Clean compiler (results for GHC are almost identical). Clean has native code compilers for MacOS, Linux and Windows;
- JMVM-C: the C++ VM version;
- JMVM-JS: the JavaScript VM version (in Safari);
- JMVM-asm.js: the asm.js VM version (in FireFox);
- Haste: the Haste Haskell to JavaScript compiler (in Safari).

All results are relative: the running times of the first implementation are divided by the running times of the second implementation. For the last column the geometric mean of all results was taken.

## 6.1 Results: native version

The JMVM version of Sapl is about 3 faster than the original version of Sapl (see Fig. 10) and therefore also much faster than other interpreters like Helium, Amanda, GHCi and Hugs. This shows

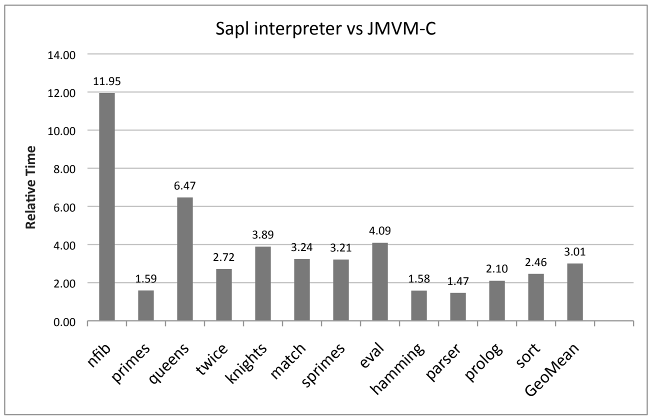

**Figure 10: Relative execution time Sapl compared to JMVM-C (JMVM-C = 1)**

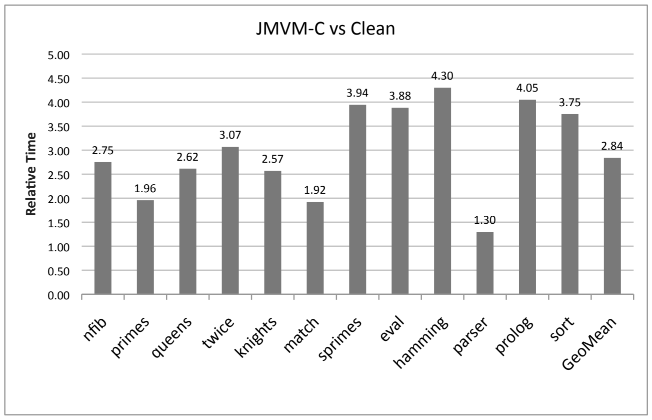

**Figure 11: Relative execution time JMVM-C compared to Clean (Clean = 1)**

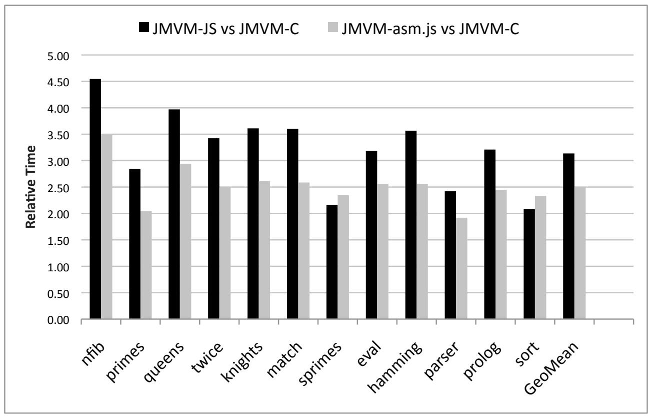

**Figure 12: Relative execution time JavaScript (Safari) and asm.js JMVM compared to JMVM-C (JMVM-C = 1)**

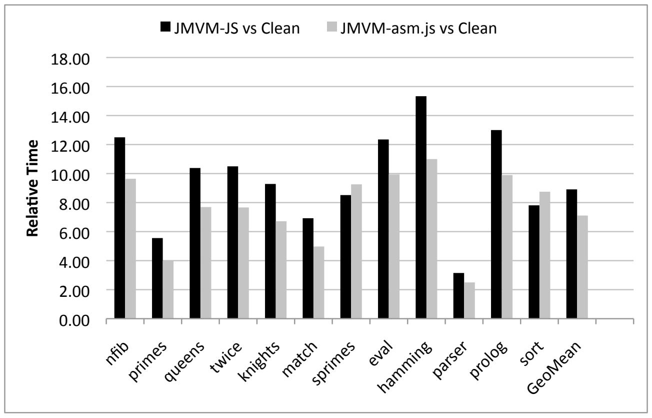

**Figure 13: Relative execution time JavaScript (Safari) and asm.js JMVM compared to Clean (Clean = 1)**

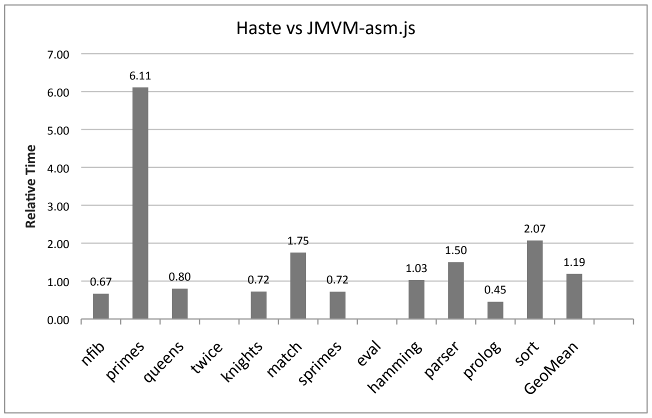

**Figure 14: Relative execution time of Haste (Safari) compared to JMVM-asm.js (JMVM = 1)**

that the virtual machine approach is probably the best choice for making a fast interpreter.

Fig. 11 shows that the JMVM version of Sapl is about 3 times slower than Clean with only a small spread between the different benchmarks (worst case 4.3 times slower). This shows that the VM approach gives a more balanced result than the graph-reduction approach that was used for the original Sapl interpreter.

*6.1.1 Results: client-side versions in JavaScript and asm.js.* The asm.js VM version of Sapl (Fig. 13) is less than 8 times slower than Clean (also with an acceptable spread, worst case 11 times slower) and about 2.5 times slower (Fig. 12) than the C++ version of the VM. Normally, a C(++) program ported to asm.js using Emscripten is less than 2 times slower than the native version. The extra performance penalty can be attributed to not using computed goto's in the JavaScript-version of JMVM. Indeed, the performance of the asm.js version is only 1.7 times slower than a C(++) version that uses a switch instead of computed goto's.

The asm.js version of the VM was run within all major web-browsers (Chrome, FireFox, Safari and Edge). The spread in the results were small, less then 10%, in most cases. Only Chrome is significantly (about 50%) slower. But JavaScript engines for the major web-browsers are frequently updated, so these figures can change rapidly.

The plain JavaScript version of the VM runs very well within Safari (only slightly slower than the asm.js version, see Fig. 12 and 13), but is much slower (2-4 times) on all other platforms.

We also ported the C(++) version of the VM to the web using Emscripten and Native Client. Native Client [9] compiles to a portable assembly format that can be run only in Chrome browsers. The Emscripten version is slightly slower (9 times slower than Clean) and the Native Client version slightly faster (6 times slower than Clean) than the asm.js version. Compiling with Emscripten results in a much larger code base (almost 10 times bigger). So using these special tools do not give a real advantage for our case.

Even the client-side asm.js version of the VM gives slightly better results (20%) than the original C-version of the Sapl interpreter and is much faster (3-4 times) than the currently used Clean → Sapl → JavaScript implementation. The JavaScript and asm.js versions of the VM also do not suffer from the stack overflow problems we encountered in the currently used JavaScript version.

*6.1.2 Results: comparison with Haste.* To make an honest comparison, the Haste benchmarks were run within Safari. At this moment the Safari JavaScript performance is better (up to 2-3 times) than for other browsers. Haste could not run the twice (crashes) and eval benchmarks (stack overflow), although Haste uses trampolining to overcome stack issues as much as possible. We faced the same problems when compiling to JavaScript in our previous attempts. For the other benchmarks (Fig. 14) we notice a great diversity in performance. Sometimes Haste is faster: nfib, queens, knights, sprimes and prolog. In other cases, JMVM is faster: primes, match, parser and sort. For hamming the results are more or less the same. Haste compiles using STG as an intermediate format. It does a better job on benchmarks that can be optimized using strictness analysis and do not consume too much memory like nfib, queens, knights and sprimes. On the other hand JMVM does a better job on higher order benchmarks and benchmarks that use a lot of memory like primes and parser. Haste relies on JavaScript garbage collection, which is not optimized for the way non-strict functional languages use memory. Although on average, JMVM and Haste are more or less on par, JMVM has a more balanced performance and does not suffer from stack overflow problems.

The code size of the run-time system of Haste and that of the generated code are up to a factor of 10 bigger than for JMVM.

## 6.2 Considerations

The slowdown factor of a VM can be defined as the difference in speed to execute a series of instructions on the VM, in comparison with executing an equivalent series of instructions natively on the same computer. We calculated the slowdown factor of JMVM by executing the nfib example. The JMVM code of nfib is almost synonym to its X86 equivalent. In turned out that the slowdown factor is 2.8. The average slowdown for our benchmark set is also 2.8. Some benchmarks perform better and some worse. For the better benchmarks this can mostly be attributed to the fact that eval is hard coded into the virtual machine. So benchmarks that create a lot of thunks and closures are expected to perform better. This is indeed the case for primes and parser, two of the better performing benchmarks. For the slower performing benchmarks one can expect that Clean did some optimisations that are not included in the Sapl to JMVM compiler. For example, Clean tries to put tuples and records on the stack instead of on the heap as much as possible. Sapl has no primitive for tuples and they are handled as an ADT. It is, however, still possible to add optimisations to the Sapl to JMVM-compiler.

*6.2.1 JavaScript vs. asm.js.* We used asm.js to obtain the best client-side performance. But for Safari the difference in performance between the JavaScript and asm.js version is less than 10%. For the other platforms (FireFox, Chrome and Edge) there is still a significant performance difference (asm.js up to twice as fast).

Because of the heavy competition in the web-browser market we expect that the performance difference between JavaScript and asm.js will also become smaller (and maybe disappear) for the other browsers in the near future. This will also make it easier to interact with JavaScript libraries from within the JMVM virtual machine.

*6.2.2 Advantages of the Virtual machine approach.* The most important advantage of the Virtual Machine approach is that it offers one execution platform that can be used both at the server- and at the client-side. In this way execution of code can easily be migrated from server and back at run-time. The client platform does not suffer from the stack overflow problems that compilation to JavaScript approaches had to deal with and the performance is still competitive in comparison with the best JavaScript compilation approaches. A program that can run in the C-version of the VM can also run in the JavaScript version of the VM with an average slowdown of 2.5. Also the code size of JMVM programs is much smaller than that of generated JavaScript (10 times or more).

*6.2.3 Comparing the Virtual machine approach with compilation to C.* For the client-side we compared the VM approach with compilation to JavaScript. It is not easy to compile Sapl to C directly in the way compilation to JavaScript was done. C does not support garbage collection like languages as JavaScript and Java do. This complicates the Sapl to C compilation considerably. It is easier to compile to Java, which is done for Frege (see Sec.7.2). An alternative way to compile to C is using the JMVM approach and to concatenate the C-code of the JMVM-instructions to one big C-function that only contains jumps for calls and branching. This removes the interpretation overhead. This works fine for small programs (30-50% faster than JMVM). But for bigger programs C compilers have serious troubles to compile the resulting C programs (taking up to 30 minutes and more).

## 7 RELATED WORK

We look at both alternative JavaScript based implementations for non-strict functional languages as well as other virtual machine based implementations for these languages.

## 7.1 Alternative JavaScript implementations of non-strict languages

In the last years many attempts were made to make efficient implementations of non-strict functional languages on top of JavaScript.

We already mentioned Haste as one of the alternatives to compile Haskell to JavaScript. Haste actually compiles STG to JavaScript and can therefore make use of almost the complete tool chain of the GHC compiler, including all the optimisations that are included in it. Haste is not intended for compiling complete Haskell programs to JavaScript, but for supporting client-server web-application development and offering easy access to existing JavaScript libraries. Haste spends large efforts to reduce the price of trampolining as much as possible, by only using it if the recursion depth is expected to be high.

Another example of a Haskell to JavaScript compiler is GHCJS. It takes a similar approach as Haste using STG as a starting point for the compilation process. GHCJS is targeted at offering the translation of the full Haskell language to JavaScript. As [5] indicates this comes with the price of relatively large JavaScript code sizes and less performance in comparison with Haste.

Another attempt to use asm.js for the implementation of a strict functional language is Slip [8]: an interpreter in asm.js for a Scheme-like language. Also for Slip a hand-coded asm.js implementation is made. To enhance this process and to maintain readability extensive use of macros is made. Like in our case, the resulting interpreter performs better than a version compiled using Emscripten.

## 7.2 Alternative virtual machines

Another VM-based implementation of a Haskell like language is Helium that uses the Lazy Virtual Machine (LVM) machine as its execution platform. Helium is a subset of Haskell, mainly used for educational purposes. The transformation of Helium to LVM proceeds in a number of steps that are similar to the steps taken in the GHC compiler. The final core language of Helium is very similar to STG. The LVM interpreter is especially targeted at this core language and less suited for executing other languages. This in contrast with JMVM that defines a more generic instruction set, also suitable for executing non-functional languages. LVM is about 2 times slower than the original Sapl interpreter [13] and therefore about 6 times slower than JMVM.

Frege [24] is an attempt to implement a Haskell like language on top of the Java Virtual machine. The Frege compiler does not generate JVM code directly, but Java code instead that is compiled to JVM by the Java compiler. The power of Frege lies in the fact that it is almost fully Haskell compatible and that it can interact with existing Java libraries. Some preliminary tests show that Frege is (much) slower than JMVM, but this can partly be caused by warm-up problems of the JVM. A Java implementation of the JMVM interpreter is almost 5 times slower than the C(++) version and therefore also slower than the JavaScript and asm.js versions.

We have chosen to implement our own virtual machine and not to make an interpreter based on an existing abstract machine like the G-machine [15], the STG-machine [20] or Clean's abc-machine [16]. The most important reason to do so is that these abstract machines are all targeted as an intermediate step in a compilation process to obtain efficient native compilers. They are not designed with the purpose to make an efficient interpreter. Also, making an efficient interpreter for these abstract machines constitutes a considerable effort, because they are more complex than JMVM.

Nevertheless, implementing an interpreter for an existing abstract machine also has advantages. There are already highly optimised compilers that generated code for these machines. Therefore, we decided to extend our efforts and to create an efficient abc-interpreter along the lines of JMVM.

## 8 CONCLUSIONS AND WAY AHEAD

In this paper we discussed a VM based implementation for the execution of non-strict functional languages. We used C(++), JavaScript and Asm.js to realize native and client-side versions of a VM that can be used to execute byte code generated by a dedicated compiler. Although our main purpose was to create a better client-side platform for the execution of non-strict functional languages, we also obtained a new efficient interpreter based platform for these languages. This platform can be used for the implementation of scripting languages or for a read-evaluate-print-loop interpreter for educational purposes. The choice of a few well chosen super instructions ensures that the interpreter has a high performance. It is better than that of existing interpreters and is only 3 times slower than Clean.

Due to the choice of the dedicated super instructions (especially the eval instruction) the Sapl to JMVM compiler can be kept simple and elegant without the need for more sophisticated optimizations. The compiler first transforms Sapl source code into a format for which the creation of thunks and closures and their evaluation is made explicit. The resulting code can now be translated to JMVM straightforwardly.

For the client-side implementation in asm.js we obtained a more predictable performance, for running non-strict FP-code in web-browsers, than for existing approaches that are based on the generation of JavaScript. All benchmarks have a performance that stays within a factor of 11 of Clean with an average of 8 times slower.

However, there is still a rather big discrepancy in performance between the native and client-side versions of the VM. This is mainly caused by the use of special techniques, like computed goto's, in the native version of the VM that cannot be used in the client-side version. But web-technology evolves and Web-Assembly [10] is a new cutting-edge web language that aims to offer a more lower level execution platform at the client-side. It can be seen as the evolution of asm.js, but offers several advantages over it, such as faster parsing, smaller code-size and faster execution. It is backed by all major browser vendors. Future releases of WebAssembly will even support tail-call optimization, which will make it a possibly suitable target for functional languages. At the moment of writing, it is still not possible to do useful experiments with WebAssembly.

We foresee several alternatives for using WebAssembly for obtaining an efficient execution platform for non-strict languages. We can use it to re-implement JMVM in the hope to obtain better performance (preferably within a factor of 5 from native Clean). We can also use it as a compilation target for Clean or Haskell directly by adding a dedicated WebAssembly back-end to their compiler chains. Both approached have their pro's and con's. Dynamic linking of extra code is easier in the first approach. The second approach will probably result in better performance. We have planned to try both.

## ACKNOWLEDGMENTS

We would like to thank László Domoszlai and Jurriën Stutterheim for the inspiring discussions we had with them on the subjects of this paper. We would also like to thank the reviewers of this paper for suggesting valuable improvements.

## REFERENCES

[1] Peter Achten, Jurriën Stutterheim, László Domoszlai, and Rinus Plasmeijer. 2014. Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*, Sam Tobin-Hochstadt (Ed.). ACM, New York, NY, USA.

[2] László Domoszlai, Eddy Bruël, and Jan Martin Jansen. 2011. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae* 3 (2011), 76–98. Issue 1. http://www.acta.sapientia.ro/acta-info/C3-1/info31-4.pdf

[3] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Editlets: Type-based, Client-side Editors for iTasks. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*, Sam Tobin-Hochstadt (Ed.). ACM, New York, NY, USA, Article 6, 13 pages. DOI: http://dx.doi.org/10.1145/2746325.2746331

[4] László Domoszlai and Rinus Plasmeijer. 2012. Compiling Haskell to JavaScript through Clean's core. In *Selected papers of 9th Joint Conference on Mathematics and Computer Science*, Vol. 36. Annales Univ. Sci. Budapest, Sect. Comp., 117–142.

[5] Anton Ekblad. 2015. *A Distributed Haskell for the Modern Web*. PhD thesis, http://ekblad.cc/lic.pdf, visited November 2016.

[6] Anton Ekblad and Koen Claessen. 2014. A Seamless, Client-centric Programming Model for Type Safe Web Applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 79–89.

[7] M. Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism* 5 (November 2003), 1–25. http://www.jilp.org/vol5/v5paper12.pdf

[8] Noah Van Es, Jens Nicolay, Quentin Stievenart, Theo D'Hondt, and Coen De Roover. 2015. *A Performant Scheme Interpreter in asm.js*. Technical Report. Software Languages Lab, Vrije Universiteit Brussel, Belgium.

[9] Google. 2016. NativeClient: Native code for web apps. (2016). https://developer.chrome.com/native-client, visited August 2016.

[10] Webassembly Community Group. 2016. WebAssembly. (2016). github.com/WebAssembly/design, visited August 2016.

[11] M. Hanus. 2007. Putting Declarative Programming into the Web: Translating Curry to JavaScript. In *Proc. of the 9th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming*. ACM Press, 155–166.

[12] D Herman, L Wagner, and Alon Zakai. 2016. ASM.JS: an extraordinarily optimizable, low-level subset of JavaScript. (2016). asmjs.org, visited November 2016.

[13] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. 2006. Efficient Interpretation by Transforming Data Types and Patterns to Functions. In *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, 19-21 April 2006, The University of Nottingham*, H. Nilsson (Ed.). Intellect.

[14] Jan Martin Jansen. 2017. The Sapl and JMVM Home Page. (2017). www.nlda-tw.nl/janmartin/sapl.

[15] Thomas Johnsson. 1984. Efficient Compilation of Lazy Evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. ACM, New York, NY, USA, 58–69.

[16] Pieter W.M. Koopman. 1990. *Functional Programs as Executable Specifications*. Ph.D. Dissertation. University of Nijmegen.

[17] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2013. The Java Virtual Machine Specification. (2013). docs.oracle.com/javase/specs/jvms/se7/html/, visited July 2016.

[18] V Nazarov, H Mackenzie, and Stegeman. 2015. GHCJS Haskell to JavaScript compiler. (2015). https:/github.com/ghcjs/ghcjs, visited November 2016.

[19] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. 2004. *An overview of the Scala programming language*. Technical Report.

[20] Simon Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming* 2, 2 (1992), 127–202.

[21] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, Freiburg, Germany, October 1-3, 2007*, N. Ramsey (Ed.). ACM, 141–152.

[22] Rinus Plasmeijer, Jan Martin Jansen, Pieter Koopman, and Peter Achten. 2008. Declarative Ajax and Client Side Evaluation of Workflows using iTasks. In *Principles and Practice of Declarative Programming, Valencia, Spain, July 2008*, Vol. PPDP 08.

[23] Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. 2014. Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In *Trends in Functional Programming*, Jurriaan Hage and Jay McCarthy (Eds.). Vol. 8843. Springer Berlin Heidelberg.

[24] Ingo Wechsung. 2016. Frege. (2016). github.com/Frege/frege, visited Juli 2016.

[25] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*. ACM, 301–312.