DRAFT

Nash: A Tracing JIT For Extension Language

Atsuro Hoshino

hoshinoatsuro@gmail.com

Abstract

This paper introduces *Nash*, a *virtual machine* (VM) for GNU Guile with tracing *just-in-time* (JIT) compiler. Nash is designed as a drop-in replacement for Guile's existing VM. Nash could be used for running script, interacted with REPL, and embedded in other programs. Design of Nash internal is discussed, including VM interpreter which records frequently executed instructions found in Guile's bytecode, and JIT compiler which emits native code from recorded instructions. Nash coexists with Guile's existing VM. Lots of Guile's features, such as bytecode interpreter, are reused in Nash. When conditions were met, Nash runs more than 40× *faster* than Guile's existing VM, without modifying the input program. Benchmark results of Nash are shown, including comparisons with other Scheme implementations.

Keywords Just-In-Time Compilation, Virtual Machine, Implementation, Scheme Programming Language

1. Introduction

From its simple design, Scheme is used for various purposes, many implementations exist. One of the use is as an extension language embedded in other program. Implementations such as Chibi-Scheme, TinyScheme are designed with use as an extension language in mind. On the other hand, there are Scheme implementations used for more expensive computations. This kind of Scheme implementations typically compiles to native code before executing. The compilation could be done *ahead-of-time* (AOT), such as in Bigloo (Serrano and Weis 1995) and Gambit (Feeley 1998), or incrementally, such as in Chez (Dybvig 2006) and Larceny (Hansen 1992), or in a mixture of AOT and JIT compilation, such as in Pycket (Bauman et al. 2015), and Racket (Flatt et al. 2013).

There exist a performance gap between Scheme implementations which does native code compilation, and implementations which doesn't. Tracing JIT compilation is a technique used in VM to improve performance by compiling the frequently executed instruction code paths. Dynamo (Bala et al. 2000) has pioneered the use of tracing JIT by tracing native code. Later the technique was used in various VMs for dynamic programming language to achieve performance improvement. Languages such as Lua (Pall 2016), JavaScript (Gal et al. 2009), and Python (Bolz et al. 2009) have made success with VMs which implement tracing JIT.

```
(define (sumv-positive vec)
  (let lp ((i 0) (acc 0))
    (if (<= (vector-length vec) i)
        acc
        (lp (+ i 1)
            (if (< 0 (vector-ref vec i))
                (+ acc (vector-ref vec i))
                acc)))))
```

Figure 1. Scheme source code of sample procedure.

Nash is a new tracing JIT VM for GNU Guile. Guile is a general-purpose Scheme implementation which could be used as an extension language, as a scripting engine, and for application development. Guile offers *libguile* to allow itself to be embedded in other program. GnuCash, gEDA, GNU Make, and GDB uses Guile as an extension language. Guile implements standard R5RS (Abelson et al. 1998), most of R6RS (Sperber et al. 2010), several SRFIs, and many extension of its own, including delimited continuation and native POSIX thread support (Galassi et al. 2002). Nash is designed to be a drop-in replacement for Guile's existing VM, which is called *VM-regular* in this paper, to achieve performance improvement.

Figure 1 shows Scheme source code of a sample procedure `sumv-positive` which contains a loop. Details of Nash internal are explained with using `sumv-positive`. The `sumv-positive` procedure takes a single argument `vec`, a vector containing numbers. The loop inside the procedure checks whether the `i`-th `vector-ref` of `vec` is greater than `0`, adds up the element if true. The loop repeat the comparison and addition with incremented `i` until `i` is greater than the `vector-length` of `vec`. In Guile, Scheme source codes are compiled to bytecode before the execution.[1] Section 2 briefly mentions some background of Guile. When Nash execute `sumv-positive`, the computation starts with bytecode interpreter. After executing the bytecode for a while, the bytecode interpreter detects a hot loop in the body of `sumv-positive`. Then the bytecode interpreter switches its state, start recording the bytecode instructions of the loop, and corresponding stack values for the instructions. Recording of instructions are described in Section 3. When the bytecode interpreter reached to the beginning of the observed loop, recorded data are passed to JIT compiler. The JIT compiler is written in Scheme, executed with VM-regular. The compiler uses recorded bytecode and stack values to emit optimized native code. The variables from the stack are used to specify types, possibly emitting *guards* to exit from the compiled native code. More the JIT compiler internals are covered in Section 4.

The rest of the sections are organized as follows. Section 5 shows results from benchmark, including comparisons between Nash and other Scheme implementations. Section 6 mentions related works. Finally, Section 7 discusses current limitations and possibilities for future work, and Section 8 concludes this paper.

2. Background

This section describes background information and brief history of tracing JIT and GNU Guile. Some of the Guile internals which relates to Nash are mentioned.

2.1 Tracing JIT

Tracing JIT is one of JIT compilation styles (Bolz et al. 2009) which assumes that:

- Programs spend most of their runtime in loops.
- Several iterations of the same loop are likely to take similar code paths.

After Dynamo (Bala et al. 2000) used the technique to trace native code, various development has been done in the area. LuaJIT is one of the successive implementation of tracing JIT VM for the Lua (Ierusalimschy et al. 1996) programming language. Pypy (Bolz et al. 2009) is a tracing JIT VM for the Python programming language. Pypy is implemented with RPython (Bolz et al. 2009) framework, which is a *meta-tracing* infrastructure to develop a tracing JIT VM by defining an interpreter of the target language. The framework was adapted to other language than Python, including Pycket (Bauman et al. 2015), a tracing JIT VM for Racket language, and Pixie, a tracing JIT VM for Pixie language, which is a dialect of Lisp.

Typical settings for tracing JIT of dynamic programming language contains an interpreter and a JIT compiler. Interpreter observes the execution of instructions, detects hot loops, and records frequently executed instructions. The recorded instructions are often called *trace*. JIT compiler then compiles the trace to get optimized native code of the hot loop. The interpreter typically has a functionality to switch between a phase for observing the loop, a phase for recording the instructions, and a phase executing the compiled native code. Compiled native code contains *guards* to terminate the execution of native code, and bring the control of program back to the interpreter. Guards are inserted when recorded trace contains conditions which might not satisfied in later iteration of the loop. For instance, the loop in Figure 1 will emit a guard which compares the value of `i` with length of the vector. In dynamic programming language such as Scheme, guard for type check may inserted as well, since compiler could generate more optimized native code when the types of the values are known at compilation time.

[1] In most case, source codes are byte-compiled before executed. Guile can run Scheme source code without compiling so that trivial computations could be done quickly.

Tag	Type	Scheme value
`XXXXXXXXXXXXX000`	heap object	‘foo, #(1 2 3) . . .
`XXXXXXXXXXXXXX10`	small integer	1, 2, 3, 4, 5 . . .
`XXXXXXXXXXXXX100`	boolean false	#f
`XXXXXXXX00001100`	character	#\ a, #\ b, . . .
`XXXXXX1100000100`	empty list	‘()
`XXXXX10000000100`	boolean true	#t

Table 1. Tag value with corresponding Scheme type and Scheme value. The “`X`” in the tag column indicates any value.

2.2 GNU Guile

GNU Guile was born to be an official extension language for GNU projects (Galassi et al. 2002). Since then, various developers have made changes to the implementation. As of version 2.1.2, Guile contains a bytecode compiler and a VM which interprets the compiled bytecode. Guile uses conservative Boehm-Demers-Weiser garbage collector (Boehm and Weiser 1988).

2.2.1 SCM Data Type

Guile’s internal data type for Scheme object is defined as typedef `SCM` in C (Galassi et al. 2002). `SCM` value contains a type tag to identify its type in Scheme, which could be categorized in two kinds: *immediates* and *heap objects*. Immediates are Scheme value with type tag, and the value to identify itself in system dependent bit size. Immediates includes booleans, characters, small integers, the empty list, the end of file object, the *unspecified* object, and *nil* object used in the Emacs-Lisp compatibility mode, and other special objects used internally. Heap objects are all the other types which could not fit itself in system dependent bit size, such as symbols, lists, vectors, strings, procedures, multiprecision integer numbers, and so on. When Guile decide the type of `SCM` value, firstly the three least significant bits

```
Disassembly of #<sumv-positive (vec)> at #x7f4ddb7fc51c:

   0    (assert-nargs-ee/locals 2 5)    ;; 7 slots
   1    (make-short-immediate 6 2)      ;; 0
   2    (vector-length 4 5)
   3    (load-u64 3 0 0)
   6    (br-if-u64-<= 4 3 #f 30)        ;; -> L5
   9    (vector-ref/immediate 2 5 0)
  10    (br-if-u64-<-scm 3 2 #t 4)      ;; -> L1
  13    (add/immediate 6 2 0)
L1:
  14    (load-u64 2 0 1)
  17    (br-if-u64-<= 4 2 #f 16)        ;; -> L4
  20    (mov 1 6)
  21    (mov 6 2)
L2:
  22    (uadd/immediate 0 6 1)
  23    (vector-ref 6 5 6)
  24    (br-if-u64-<-scm 3 6 #t 4)      ;; -> L3
  27    (add 1 1 6)
L3:
  28    (br-if-u64-<= 4 0 #f 9)         ;; -> L6
  31    (mov 6 0)
  32    (br -10)                        ;; -> L2
L4:
  33    (mov 0 2)
  34    (mov 1 6)
  35    (br 2)                          ;; -> L6
L5:
  36    (mov 1 6)
L6:
  37    (mov 5 1)
  38    (return-values 2)               ;; 1 value
```

Figure 2. Byte compiled code of `sumv-positive`. The contents is slightly modified from output of disassembler for displaying purpose.

(called *tc3* tag in Guile) of `SCM` value is used to decide whether the value belongs to immediates or heap objects. Table 1 shows the type tags for immediates and heap objects. When tc3 tag was `000`, the value belongs to heap objects. All the other values of tc3 tag are immediates, though some of the tag values are unused. For instance, tc3 tag `010` and `110` are used for small integer. For small integer, only the first two bits are used to identify the type, the rest of the bits are used to contain the integer value. For instance, Scheme value 0 is SCM 00000010, Scheme value 1 is SCM 00000110, Scheme value 2 is SCM 00001010, and so on. Types of various heap object are decided by using the rest of `SCM` value, possibly referencing the address derived from non-tc3 tag value in `SCM`.

2.2.2 Bytecode Compiler

Guile's bytecode compiler is designed as *compiler tower*. The compiler consists from several compilers defining tower of languages. Each step of the compilation sequence knows how to compile down to the step below, until the compiled output turns into bytecode instruction set executed by the VM.

In Guile version 2.1.2, Scheme input program is first translated to a program in *tree-il* language, an internal representation used by Guile. Then resulting tree-il program is compiled to *cps*, which is another internal representation, then the resulting cps code is translated to bytecode. Guile contains compilers for Emacs-Lisp and Ecmascript, which compiles to tree-il. Compilation of tree-il results from Emacs-Lisp and Ecmascript could reuse the compilation from tree-il to bytecode used for Scheme. The definition of the compilation steps could be modified, which helps the user to add a new high-level language compiled to `tree-il`, or directly compiling to bytecode, or compiling to different new target. Guile's bytecode compiler applies various optimizations, which all of them are turned on by default. The optimizations could be turned off to save compilation time, with sacrificing some run time performances.

Figure 2 shows compiled bytecode of `sumv-positive`. Each bytecode instruction takes arguments and its use varies, some arguments are used as constant, some are used as an index value to read or write a value in current stack. The first line of the figure shows the procedure name and memory address of the byte-compiled data of `sumv-positive`. The numbers shown in the left of each line are bytecode *instruction pointer* (IP) offset. For IP offsets specified as jump destination, a label starting from `L` is shown (IP offset 14, 22, 28, 33, 36, and 37 in the figure). The bytecode possibly causing a jump contains a comment with the destination label (IP offset 6, 10, 17, 24, 28, 32 and 35 in the figure).

When the procedure `sumv-positive` is called, VM-regular start executing the bytecode from IP offset 0. Later, the execution reaches to IP offset 32, `(br -10)`. The `br` bytecode instruction is unconditional jump to specified IP offset, which is -10 in the figure. This is a backward jump to IP offset 22 (labeled as `L2` in the figure), which indicates a loop. In the bytecode instruction inside the loop, IP offset 24 contains a branching instruction, which may skip `(add 1 1 6)` instruction in IP offset 27. The loop contains a jump to `L6` in IP offset 28, to exit from the loop and return the value from `sumv-positive` procedure to the caller.

The bytecode was compiled with optimizations turned on. The compiler has moved `vector-length` out of the loop, by loop-invariant code motion. The bytecode compiler also refactored the two calls to `vector-ref`, by common sub-expressions elimination.

2.2.3 VM Engine

Guile uses C function to interpret compiled bytecode. This C function is called *VM engine* in Guile. VM engine is defined in a dedicated file named `vm-engine.c`, which is included multiple times from other source code. Figure 3 shows the snippets of `vm-engine.c`. In existing implementation, Guile contains two VM engines: one is *vm-regular-*

```
static SCM
VM_NAME (scm_i_thread *thread, struct scm_vm *vp,
         scm_i_jmp_buf *registers, int resume)
{
  ...
  VM_DEFINE_OP (33, br, ...)
    {
      ...
      NEXT (offset);
    }
  ...
  VM_DEFINE_OP (87, add_immediate, ...)
    {
      ...
      NEXT (1);
    }
  ...
  VM_DEFINE_OP (152, uadd_immediate, ...)
    {
      ...
      NEXT (1);
    }
  ...
}
```

Figure 3. VM_NAME defined in C source code.

engine, the engine used by VM-regular, and another is *vm-debug-engine*, which is used for debugging and contains more functionality to help developers during the debug. vm-debug-engine can invoke user defined hook procedures in several predefined places, such as before a procedure call, before returning values, and before executing each bytecode instruction. The invocation of hooks are disabled in vm-regular-engine for performance reason. When including vm-engine.c, VM_NAME is defined with different literal. i.e.: Some C codes resembling below are written in other file than vm-engine.c:

```
#define VM_NAME vm_regular_engine
#define VM_USE_HOOKS 0
#include "vm-engine.c"
#undef VM_NAME
...
#define VM_NAME vm_debug_engine
#define VM_USE_HOOKS 1
#include "vm-engine.c"
#undef VM_NAME
```

Inside the VM_NAME function, each bytecode instructions is defined with VM_DEFINE_OP macro with unique instruction number, with NEXT at the last of definition body to perform next instruction. VM_DEFINE_OP macro fills in the jump table used by VM_NAME to define jump destinations.[2]

[2] Strictly speaking, Guile chooses jump table or `switch ... case` expression at build time for dispatching bytecode, by deciding whether the platform supports label as values (computed goto).

```
union scm_vm_stack_element
{
  scm_t_uintptr as_uint;
  scm_t_uint32 *as_ip;
  SCM as_scm;
  double as_f64;
  scm_t_uint64 as_u64;
  scm_t_int64 as_s64;
  ...
};
```

Figure 4. C code defining union of stack element in VM engine.

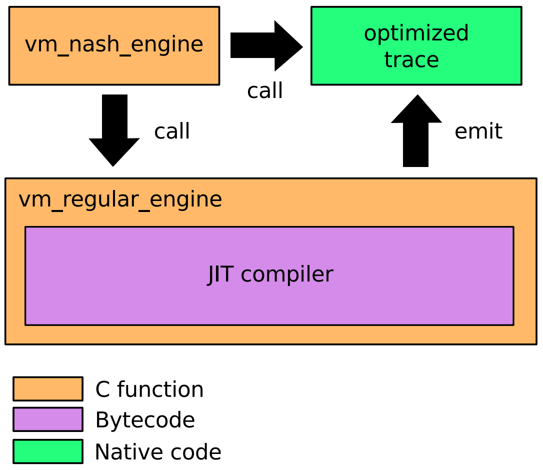

Figure 5. A diagram showing software components and language its written.

Some of the instructions continue the interpretation with NEXT using constant value, such as add_immediate and uadd_immediate which using 1, some with using offset value specified in argument, such as br.

The stack element type used in VM_NAME is defined as C union, shown in Figure 4. The actual type used by each bytecode instruction differs, some of the instructions use stack element as SCM, or as scm_t_uint64 which is an internal type for unsigned 64 bit in Guile. For instance, bytecode instruction add_immediate adds a constant immediate value to a stack element specified by given index, with referring the stack element as SCM. Bytecode instruction uadd_immediate does almost the same, except for treating the stack element as scm_t_uint64 type. This type specializations are done at the time of bytecode compilation, to remove unnecessary tagging and untagging of SCM values.

3. Nash Interpreter

3.1 Nash Overview

Nash is designed as a drop-in replacement of VM-regular, could be used to run a script, has REPL, and could be embedded in a C program as extension language. Figure 5

```
static SCM
VM_NAME (scm_i_thread *thread, struct scm_vm *vp,
         scm_i_jmp_buf *registers, int resume)
{
  ...
  VM_DEFINE_OP (1, call, ...)
    {
      ...
      VM_NASH_CALL (old_ip);
    }
  ...
  VM_DEFINE_OP (3, tail_call ...)
    {
      ...
      VM_NASH_TAIL_CALL (old_ip);
    }
  ...
  VM_DEFINE_OP (33, br, ...)
    {
      ...
      VM_NASH_JUMP (offset);
    }
  ...
  VM_DEFINE_OP (152, uadd_immediate, ...)
    {
      ...
      NEXT (1);
    }
  ...
}
```

Figure 6. Modified VM_NAME function shown in Figure 3.

contains a diagram showing which software components are written as C function, compiled bytecode, or JIT compiled native code. C function vm_nash_engine is the interpreter used by Nash, which does the bytecode interpretation to count loops, records the instructions in loops, and calls compiled native code. When traces and stack values are recorded, vm_nash_engine calls vm_regular_engine, the C function used by VM-regular, and pass the recorded trace and stack values. The bytecode of compiler interpreted by vm_regular_engine is written in Scheme. After successful compilation, vm_regular_engine emits a native code of the input trace. The native code is called from vm_nash_engine when the same loop was encountered.

3.2 VM Engine For Nash

The bytecode interpreter in Nash uses the vm-engine.c file, which was used for defining vm engines as described in Section 2.2.3. The file vm-engine.c is included once more with similar code to below:

```
#define VM_NAME vm_nash_engine
#define VM_NASH 1
#include "vm-engine.c"
#undef VM_NAME
```

Few small modifications were made to vm-engine.c. Nash adds two kinds of macros to mark interpreter, one for recording instructions, and others to detect hot loops and execute compiled native codes. A C macro VM_NASH_MERGE is used for recording, and three C macros VM_NASH_JUMP, VM_NASH_CALL, and VM_NASH_TCALL are used for detecting hot loops and entering compiled native code.

3.2.1 Finding Loops

Figure 6 shows a snippet containing modifications made to VM_NAME. VM_NASH_JUMP is used in definition body of br, VM_NASH_CALL in call, and VM_NASH_TAIL_CALL in tail-call. Bytecode definitions br, call, and tail-call are marked since these bytecode perform a jump, which may start a loop.[3] Bytecode definitions which do not start a loop, such as uadd_immediate, are unmodified. The definition of br contains VM_NASH_JUMP with a parameter offset at the end of definition body. When offset was negative, it means a backward jump which is detected as a loop by Nash. When br with negative offset was found in interpreted bytecode, vm_nash_engine looks for a native code with the next IP. Similarly, VM_NASH_CALL and VM_NASH_TAIL_CALL use its argument old_ip to detect loop from consequent calls to identical IP.

If a native code was found, the native code is executed. If not, vm_nash_engine increment the counter value for the IP, and if the counter value exceeds a threshold parameter, vm_nash_engine starts recording the bytecode instruction in current loop. For instance, vm_nash_engine can find the loop in sumv-positive from the bytecode (br -10) in IP 32 of Figure 2.

Codes used internally in VM_NASH_JUMP, VM_NASH_CALL, and VM_NASH_TCALL are mostly shared. One of the differences between these three macros is to use different strategy to decide loops as hot, by using different values to increment loop counter. For instance, VM_NASH_JUMP may add 2 to the loop counter, while VM_NASH_CALL may add 1, which will result in a setting that backward jumps get hot sooner than consequent calls. VM_NASH_JUMP, VM_NASH_CALL, and VM_NASH_TACLL are defined as NEXT when including the file vm-engine.c to define other VM engines.

3.2.2 Recording Instructions

Figure 7 shows dumped sample data of recorded bytecode and stack values, and Figure 8 shows the modified contents of NEXT. When vm_nash_engine found a loop, observed bytecode and stack values are recorded by VM_NASH_MERGE. Figure 8 shows how the interpretation continues with updating the value of ip, which is a variable in VM_NAME used for bytecode IP. When the value of ip match with beginning of the loop, VM_NASH_MERGE will stop the recording, and pass

[3] Guile has more bytecode instructions for branching, such as br-if-u64-<=. These branching instructions are marked with VM_NASH_JUMP since they may perform a backward jump, though not shown in the figure.

```
7f4ddb7fc574  (uadd/immediate 0 6 1)              ; #(#x3e #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x3e)
7f4ddb7fc578  (vector-ref 6 5 6)                  ; #(#x3f #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x3e)
7f4ddb7fc57c  (br-if-u64-<-scm 3 6 #t 4)          ; #(#x3f #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc588  (add 1 1 6)                         ; #(#x3f #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc58c  (br-if-u64-<= 4 0 #f 9)             ; #(#x3f #x2ba #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc598  (mov 6 0)                           ; #(#x3f #x2ba #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc59c  (br -10)                            ; #(#x3f #x2ba #x1 #x0 #x3e8 #x7f4ddb808660 #x3f)
```

Figure 7. Bytecode instructions and stack values recorded with running `sumv-positive`.

```
# define NEXT(n)                                  \
  do                                              \
    {                                             \
      ip += n;                                    \
      VM_NASH_MERGE ();                           \
      ...
      op = *ip;                                   \
      goto *jump_table[op & 0xff];                \
    }                                             \
  while (0)
```

Figure 8. Modified definition of `NEXT`.

the recorded data to JIT compiler. The recorded bytecode instructions and stack values in Figure 7 were made by running `sumv-positive` with passing a length 1000 `vector` containing random small integer numbers from -10 to 10 as argument. The hexadecimal numbers in left are the absolute bytecode IP of each bytecode instruction, and the commented out vector in each line contains SCM representation of the values in the stack at the time of recording. The first bytecode `(uadd/immediate 0 6 1)` in Figure 7 is shown at IP offset 22 in Figure 2. `vm_nash_engine` continued the recording with IP offset 23, 24, 27, 28, and 31. Then at IP offset 32, the last bytecode `(br -10)` is recorded and jumped back to IP offset 22, which is labeled as `L2` in Figure 2. The bytecode IP matched with the IP where the recording started, `vm_nash_engine` stopped the recording. `VM_NASH_MERGE` is defined with empty body when including `vm-engine.c` file for other VM engines.

4. Nash Compiler

This section describes the details of JIT compiler in Nash, which is written in Scheme. Compiled bytecode of the JIT compiler is executed by `vm_regular_engine`.

4.1 Trace To IR

Nash compiles traces to relaxed A-normal form (Flanagan et al. 1993) internal representation (IR) before assigning registers and assembling to native code. Figure 9 shows IR of primitive operations compiled from the recorded trace in Figure 7. The IR primitives contains two `lambda` terms, the first block is for prologue, and the second block is for loop body. The IR uses `let*` instead of `let` to express the sequence of computation. Each primitive operation takes two

```
(lambda ()
  (let* ((_    (%snap 0))
         (v0   (%sref 0 #f))
         (v1   (%sref 1 1))
         (v3   (%sref 3 67108864))
         (v4   (%sref 4 67108864))
         (v5   (%sref 5 131072))
         (v6   (%sref 6 67108864)))
    (loop v0 v1 v3 v4 v5 v6)))
(lambda (v0 v1 v3 v4 v5 v6)
  (let* ((v0   (%add v6 1))
         (_    (%snap 1 v0 v1 v6))
         (r2   (%cref v5 0))
         (r2   (%rsh r2 8))
         (_    (%lt v6 r2))
         (r2   (%add v6 1))
         (v6   (%cref v5 r2))
         (_    (%snap 2 v0 v1 v6))
         (_    (%typeq v6 1))
         (_    _)
         (r2   (%rsh v6 2))
         (_    (%lt v3 r2))
         (_    (%snap 3 v0 v1 v6))
         (v1   (%addov v1 v6))
         (v1   (%sub v1 2))
         (_    (%snap 4 v0 v1 v6))
         (_    (%gt v4 v0))
         (v6   v0))
    (loop v0 v1 v3 v4 v5 v6)))
```

Figure 9. IR of recorded trace in relaxed A-normal form.

arguments, except for `%snap` operation. Primitive operations updating a variable, such as `%add`, has the variable on the left side of expression, which is updated by the corresponding operation. Primitive operations without variable update contain a symbol `_` at the left. The variables starting with the letter `v` indicates that the variable is loaded from current stack. The variables starting with the letter `r` indicates that the variable is for temporal use only.

4.1.1 Snapshot

Between recorded bytecode instructions, `%snap` expressions may inserted to make *snapshot* data. Snapshot contains various information to recover the state of `vm_nash_engine` when native code passed the control back. Snapshot data includes local indices to store variables, and a bytecode IP where the interpretation continue. The expression `%snap` takes variable number of arguments: the first argument is a *snapshot ID*, which is a unique integer number to identify the snapshot in single trace. The rest of the arguments are local variables to be stored to current stack. In Figure 9, Nash inserted `%snap` expression at the beginning, and before the primitive operations `%lt`, `%typeq`, `%addov`, and `%gt`, which act as guard. The primitives `%lt` and `%gt` does arithmetic less-than and greater-than comparisons, respectively, the primitive `%typeq` does type check with given variable and type, and the primitive `%addov` does addition with overflow check. When the result of guard differed from the result observed at the time of JIT compilation, native code execute the recovering steps to setup the state in `vm_nash_engine`, and input program continues with bytecode interpreter.

4.1.2 Prologue section

The prologue section, the first `lambda` block shown in Figure 9, loads initial values from the stack with `%sref` primitive. The first argument number passed to `%sref` is a local index offset, the second argument is an integer used internally to represent the type of expected local in the stack. For instance, the value `1` is for `fixnum`, which means small integer value in Scheme, `131072` is used for vector object, `67108864` is for `u64`, which is a non-SCM unsigned 64 bit integer value to alias `scm_t_uint64` shown in Figure 4, and so on.

Type information are specified from bytecode operation, or from the stack values recorded alongside with bytecode instructions. The tc3 tag, described in Section 2.2.1, of each value is observed and type check for the locals are added when necessary. For instance, in the line containing `(add 1 1 6)` in Figure 7, the second element in the stack is `#x2a2`, which is `1010100010` in binary. Nash could decide this value as a small integer, since it has tc3 tag `010`.

The value `#f` in `%sref` primitive means that there is no need for type check, for instance the local is overwritten without referencing. The variable `v0`, which hold local `0` in above example is immediately overwritten by result of `%add` primitive with the first line of loop body. There is not need to load this local from current stack, though such dead-code eliminations are not yet implemented.

4.1.3 Loop body section

The loop body section, the second `lambda` block in Figure 9, is compiled by translating each recorded bytecode instruction sequentially.

uadd/immediate The first primitive operation contains `%add`, which does arithmetic addition with variable `v6` and constant value 1. The result of addition overwrites variable `v0`. No overflow check is done with `uadd/immediate` in bytecode interpreters, and result will wrap around. Native code followed this behavior.

vector-ref Then a snapshot 1 is inserted, and the primitive operations for `vector-ref` follows. The primitive operations contain vector index range check, by comparing the length of vector with the index value passed to `vector-ref` instruction. For Scheme `vector` object, Guile uses the first one word to store a *tc7* tag and the length of the vector. A tc7 tag is, like tc3 tag, a 7 bits long tag value used to distinguish types. The length is left shifted for 8 bits so that tc7 tag and the length could fit in single word. Actual vector elements are stored from the memory address of the SCM object plus one word.

The primitive operation `%cref` in line 13 loads a value from Scheme heap object with offset 0, then store the loaded value to temporary register `r2`. The `r2` is passed to `%rsh` in line 14, which does arithmetic right shift for 8 bits and overwrite the value of `r2`. Now the variable `r2` contains the reproduced vector length, and compared with `v6`, which is the variable holding local 6, which is the index value used in `vector-ref` bytecode instruction of recorded trace. Line 16 adds `1` to `v6` get offset of vector element. Line 17 does another `%cref` to load the vector element, and overwrites the value of `v6`.

br-if-u64-<-scm Line 18 contains a snapshot used by next primitive operation `%typeq`, which does a type check of `v6` with `fixnum`. The variable `v6` is the value loaded from `vector`, which was observed as `fixnum` at the time of JIT compilation. Line 20 shows empty value assigned to empty value. This line used to contain a `%snap` expression, though the JIT compiler has optimized away the snapshot, since the bytecode IP destination in snapshot data was identical to the previous snapshot. Nash does few on-the-fly optimizations, such as this cached snapshot reuse, duplicated guard elimination, and constant folding. Variable `v6` is right shifted for 2 bits to move away the tc3 tag of `fixnum`, and the result is stored to variable `r2` in line 21, to compare with variable `v3` shown in line 22. Bytecode instruction `br-if-u64-<-scm` takes `u64` type as first argument, SCM type as second argument, and compares the two. The type of variable `v3` is determined at compile time as `u64`, no type check is done.

add Line 23 contains a snapshot, which is used when arithmetic overflow occurred. Line 24 adds two `fixnum` values in `v1` and `v6` with `%addov` primitive, and overwrites the contents of `v1`. Type checks for `v1` and `v6` are not done, since the `fixnum` type check done with `%typeq` for `v6` is still valid, and `fixnum` type check for `v1` is already done in prologue section. Resulting type from addition of two `fixnum` is again a `fixnum` unless it overflows, thus type check for `v1` inside loop body is eliminated. The `%sub` primitive in line 25 moves away the extra tc2 bits added by `%addov`.

br-if-u64-<= Line 26 inserts another snapshot. Then in line 27, two 64 bit unsigned values in `v4` and `v0` are compared. Types of variables are determined from the bytecode.

```
----      [snap  0] ()
0001      (%sref    r14 +0 ---)
0002      (%sref    r15 +1 fixn)
0003      (%sref    r9 +3 u64)
0004      (%sref    r8 +4 u64)
0005      (%sref    rcx +5 vect)
0006      (%sref    rdx +6 u64)
==== loop:
0007      (%add     r14 rdx +1)
----      [snap  1] ((0 u64) (1 fixn) (6 u64))
0009      (%cref    r11 rcx +0)
0010      (%rsh     r11 r11 +8)
0011    > (%lt      rdx r11)
0012      (%add     r11 rdx +1)
0013      (%cref    rdx rcx r11)
----      [snap  2] ((0 u64) (1 fixn) (6 scm))
0015    > (%typeq   rdx fixn)
0016      (%rsh     r11 rdx +2)
0017    > (%lt      r9 r11)
----      [snap  3] ((0 u64) (1 fixn) (6 scm))
0019      (%addov   r15 r15 rdx)
0020      (%sub     r15 r15 +2)
----      [snap  4] ((0 u64) (1 fixn) (6 scm))
0022    > (%gt      r8 r14)
0023      (%move    rdx r14)
```

Figure 10. Primitive operation of recorded trace under x86-64 architecture. Slightly modified from dumped output for displaying purpose.

mov Line 28 simply does a move, and overwrites the contents of `v6` with `v0`.

br Then the IR shows a call to `loop`, which tells that the computation jumps to the beginning of loop body section. Loop body of native code exit when any of the guards failed. For instance, when the result returned by `(%gt v4 v0)` differed from the result observed at the time of JIT compilation, native code will pass the control back to `vm_nash_engine`, recover the interpreter state by using snapshot data from `(%snap 3 v0 v1 v6)` and the bytecode interpretation of input program will continue from bytecode IP `7f4ddb7fc5b0`, which is the jump destination of `(br-if-u64-<= 4 0 #f 9)` in recorded trace. [4]

4.2 IR To Native Code

Figure 10 shows IR after register assignment. The numbers in left of each line, which are omitted for snapshot data, tells primitive operation number. The variables for primitive operations in Figure 10 are using register names in x86-64 architecture. Nash uses a module defining architecture specific register variables. At the time of writing, x86-64 implementation is the only one which exist.

[4] `7f4ddb7fc5b0 = 7f4ddb7fc8c + (9 * 4)`. `7f4ddb7fc8c` is the absolute bytecode IP of `(br-if-u64-<= 4 0 #f 9)` shown in recorded trace, 9 is the offset argument passed to `br-if-u64-<=`, and 4 is the size of single byte used for bytecode.

```
0x02cc005b mov    r14,QWORD PTR [rbx]
0x02cc005e mov    r15,QWORD PTR [rbx+0x8]
0x02cc0062 mov    r9,QWORD PTR [rbx+0x18]
0x02cc0066 mov    r8,QWORD PTR [rbx+0x20]
0x02cc006a mov    rcx,QWORD PTR [rbx+0x28]
0x02cc006e mov    rdx,QWORD PTR [rbx+0x30]
0x02cc0072 nop    WORD PTR [rax+rax*1+0x0]
loop:
0x02cc0078 lea    r14,[rdx+0x1]
0x02cc007c mov    r11,QWORD PTR [rcx]
0x02cc007f sar    r11,0x8
0x02cc0083 cmp    rdx,r11
0x02cc0086 jge    0x02ccc028    ->1
0x02cc008c lea    r11,[rdx+0x1]
0x02cc0090 lea    rax,[r11*8+0x0]
0x02cc0098 mov    rdx,QWORD PTR [rax+rcx*1]
0x02cc009c test   rdx,0x2
0x02cc00a3 je     0x02ccc030    ->2
0x02cc00a9 mov    r11,rdx
0x02cc00ac sar    r11,0x2
0x02cc00b0 cmp    r9,r11
0x02cc00b3 jge    0x02ccc030    ->2
0x02cc00b9 mov    r11,r15
0x02cc00bc add    r11,rdx
0x02cc00bf jo     0x02ccc038    ->3
0x02cc00c5 mov    r15,r11
0x02cc00c8 sub    r15,0x2
0x02cc00cc cmp    r8,r14
0x02cc00cf jle    0x02ccc040    ->4
0x02cc00d5 mov    rdx,r14
0x02cc00d8 jmp    0x02cc0078    ->loop
0x02cc00dd nop    DWORD PTR [rax]
```

Figure 11. Native code compiled from trace, under x86-64 architecture.

Nash uses GNU lightning as an assembler backend. GNU lightning is a JIT compilation library, which runs under various architectures including aarch64, alpha, arm, ia64, mips, powerpc, s390, sparc, and x86. Nash contains a thin C wrapper which binds SCM type to the types understood by GNU lightning. The resulting bindings are called from Scheme code under `vm_regular_engine` just like other Scheme procedures defined in C.

Figure 11 shows dumped native code of primitive operations shown in Figure 10 under x86-64 architecture. The native code contains a mark to show the beginning of loop body with `loop:`, and corresponding snapshot ID number for jump instructions.

Native code compiled in Nash contains debug symbol to interact with JIT compilation interface in GDB (Stallman et al. 2002), a well-known open-source debugger. This debugging support is turned off by default, turned on by pass-

Benchmark name	Nash	Racket	Pycket
sumfp	0.021	0.127	0.015
mbrot	0.036	0.122	0.018
sum	0.118	0.373	0.088
ctak	0.844	1.050	0.091
fibc	1.282	1.116	0.095
paraffins	1.160	0.725	1.500
parsing	1.886	0.149	0.154
dynamic	4.526	0.549	2.043
Geometric mean	0.420	0.312	0.235

Table 2. Selected benchmark results and geometric mean in Figure 12 shown in numbers. Smaller is better.

ing command line option when invoking `guile` executable. [5]

5. Evaluation

5.1 Settings

Performance of Nash is evaluated with modified cross platform benchmark suite from Pycket project. The source code of the benchmarks originate from Larceny and Gambit project. Some modifications were done to the Pycket version for evaluating Nash. Benchmarks `bv2string`, `ntakl` and `quicksort` were added. All of the three newly added benchmarks exist in the original Larceny and Gambit benchmark suite. Iteration counts for `ctak`, `fft`, `pnpoly`, `fibc`, and `ray` are decreased, which were taking long execution time in Guile's VM-regular. The modified benchmark suite contains 57 programs.

The benchmark results were taken under a machine with Intel Core-i5 3427-U and 8GB of memory, running Arch Linux, Linux kernel 4.5.4. The Scheme implementations and its versions used are Guile VM-regular version 2.1.2, Racket version 6.5, and Pycket git revision 5f98bfe (with RPython hg revision 83529:2179c). Total elapsed time of each program including JIT warm up time was measured.

5.2 Results

Figure 12 shows relative total time of each benchmark result and geometric mean normalized to Guile's VM-regular. The range of the plot in the figure is limited from 0 to 2 for displaying purpose. Table 2 contains selected benchmark results and geometric mean shown in numbers. Geometric means of 57 programs shows that Nash runs approximately 2.4× *faster* than VM-regular, 1.4× *slower* than Racket, and 1.8× *slower* than Pycket.

Some of the benchmarks have shown significant performance improvement in Nash compared to VM-regular.

[5] Other than debug symbol, Nash contains a command line option to dump intermediate data used during compilation. Contents of Figure 7, 9, 10, and 11 are obtained from the dumped output.

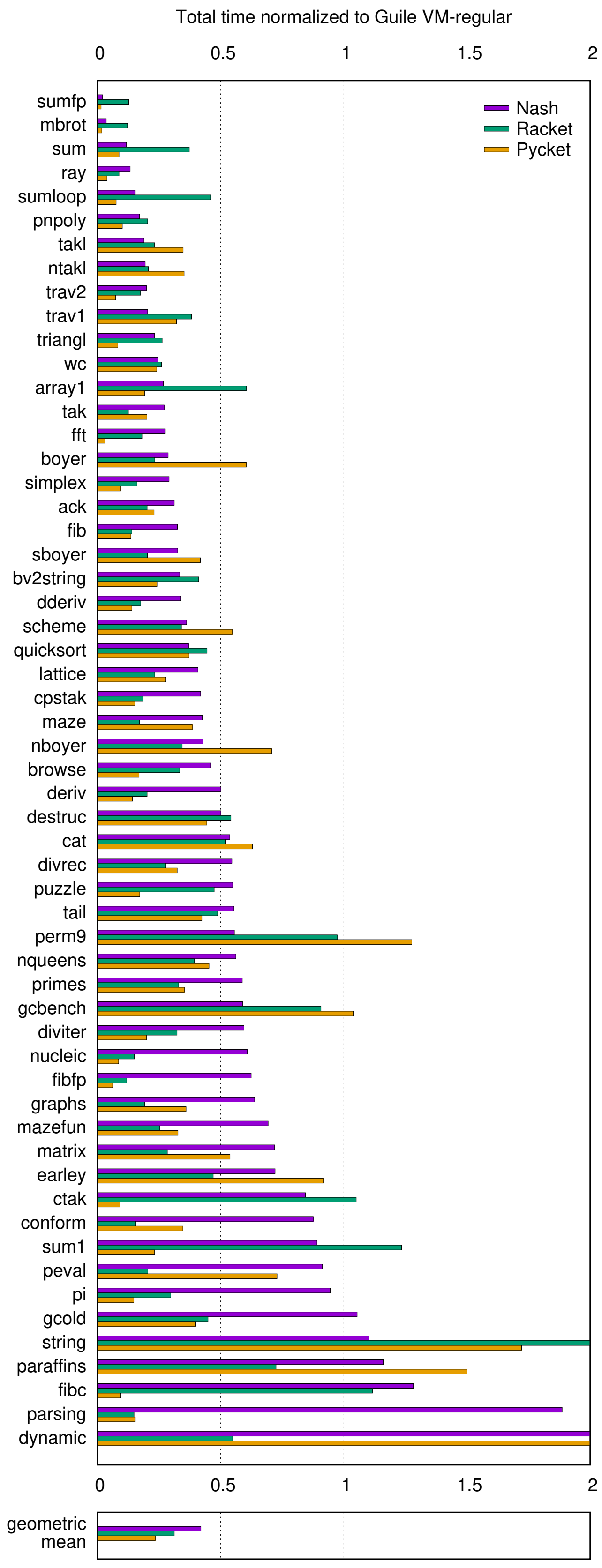

Figure 12. Relative total time of benchmarks and geometric mean normalized to Guile VM-regular. Smaller is better.

Benchmark `sumfp` and `mbrot` contain tight loop with Scheme `flonum`, which is implemented with heap object in Guile. When possible, Nash tries to unbox `flonum` values and keep using the unboxed value in native code. Nash runs `sumfp` $47\times$ faster, and `mbrot` $27\times$ faster than VM-regular. The results of `sumfp` and `mbrot` are about $6\times$ and $3.3\times$ faster than Racket, respectively. However, both benchmarks still run slower than Pycket.

Some of the benchmarks performed worse than Guile's VM-regular. Benchmark `parsing` and `dynamic` contain program to parse a Scheme source code. These programs include lots of branching conditions. The `parsing` and `dynamic` performed approximately $1.9\times$ and $4.5\times$ slower than VM-regular, respectively.

6. Related Work

Implementation of Nash was inspired from pioneers in tracing JIT field. Nash does type checks in VM interpreter before running compiled native code, which was done in TraceMonkey (Gal et al. 2009) with similar design.

The mechanism in interpreter to record instructions and detect hot loops are inspired from Pypy and RPython (Bolz et al. 2009). The way how Nash mark `NEXT` and loop starting bytecode definitions are influenced by the approach used in interpreter written with RPython.

Use of A-normal form (Flanagan et al. 1993) was inspired from Pycket (Bauman et al. 2015). Though Pycket expand the source code and generates JSON data, and parses it to AST for execution. Both Nash and Pycket use tracing JIT in its implementation. Both performed well with benchmarks containing tight loop, such as `sumfp`, `sum`, and `sumloop`. Pycket performed better than Nash. From Table 2, Pycket performed well with benchmark containing extensive use of `call/cc`, such as `fibc` and `ctak`. Nash implements `call/cc` with C stack, to keep compatibility with C interface of libguile, which did not led to performance improvement. Both performed bad in benchmarks containing lots of condition branches, such as `paraffins` and `dynamic`. Nash performed bad in `parsing` benchmark also, but the performance of `parsing` in Pycket was close to Racket.

Design of snapshot was inspired from LuaJIT (Pall 2009). LuaJIT uses more sophisticated approach and compressed snapshot data. LuaJIT used NaN-tagging, which enables efficient handling of unboxed floating point numbers. Guile once had an attempt to use NaN-tagging (Wingo 2011), though the attempt wasn't merged to Guile source code, due to supporting conservative garbage collector under 32 bit architectures.

7. Limitations and Future Work

Nash is still an experimental implementation, still has a lot of space for improvement. Some of the possibilities for future works follows.

Support more bytecode instructions Nash still have bytecode instructions which are not implemented at all, such as `prompt` and `abort` used for delimited continuation, or partially implemented, such as arithmetic operations. Arithmetic operations for multi-precision numbers and complex numbers are not yet implemented. When JIT compiler encountered unsupported bytecode, the abort the work and fall backs to `vm_nash_engine`.

Detect more loops Nash detects loops with backward jump, tail-calls, and consequent calls for non-tail-call recursion (*down-recursion*[6]), though not with consequent returns from non-tail-call recursions (*up-recursion*) yet. Nash does not detect *looping side traces*. Side trace start from frequently taken exit in native code. Side traces are usually patched to existing trace when `vm_nash_engine` encountered beginning bytecode IP of existing trace. In looping side trace, bytecode instructions loops inside the side trace instead of patched back.

Optimize IR Nash does only a few IR level optimizations. In prologue section of IR shown in Section 4.1.2, unnecessary load from stack exist, which could be omitted by dead-code elimination. More optimizations could be done, such as loop invariant code motion, escape analysis, and so on. Also, Nash currently uses naive method to assign registers. More sophisticated method such as Linear-Scan register allocation (Poletto and Sarkar 1999) could be used.

Blacklist Benchmarks such as `parsing` and `dynamic` were slower than Guile VM-regular. Programs containing large number of branching conditions need some treatments to perform well under tracing JIT. One approach is to limit the JIT compilation when branching exceed certain threshold. These thresholds mechanism are sometime called *blacklisting* of trace. Nash could take more sophisticated approach to blacklist unwanted traces.

8. Conclusion

This paper has shown how Nash is designed. Nash reused existing bytecode interpreter in Guile, turned it to a trace recording interpreter with few small modifications. Nash coexist with existing VM, which is used for JIT compiler written in Scheme. Existing bytecode compiled by Guile are traced, recorded and compiled to native code. Performance comparison between Nash, Guile's existing VM, Racket and Pycket were done. With keeping itself as an extension language, Nash showed significant speed ups in programs with tight loops compared to Guile's existing VM, and close performance with Scheme implementation with native compiler in several of the benchmarks.

[6] Consequent calls to procedure are called as down-recursion in Nash, because Guile's stack grows down

Acknowledgments

Nash exist because of the hard works done for GNU Guile by many open source developers.

References

H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr, D. H. Bartley, R. Halstead, et al. Revised5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11 (1):7–105, 1998.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A tracing jit for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 22–34. ACM, 2015.

H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9): 807–820, 1988.

C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

R. K. Dybvig. The development of chez scheme. In *ACM SIGPLAN Notices*, volume 41, pages 1–12. ACM, 2006.

M. Feeley. Gambit-c version 3.0. *An implementation of Scheme available via http://www. iro. umontreal. ca/˜ gambit*, 6, 1998.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

M. Flatt et al. The racket reference, 2013.

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

M. Galassi, J. Blandy, G. Houston, T. Pierce, N. Jerram, and M. Grabmller. Guile reference manual, 2002.

L. T. Hansen. *The impact of programming style on the performance of Scheme programs*. PhD thesis, Citeseer, 1992.

R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6): 635–652, 1996.

M. Pall. Luajit 2.0 intellectual property disclosure and research oppotunities. http://lua-users.org/lists/lua-l/2009-11/msg00089.html, 2009. [Online, accessed June 14, 2016].

M. Pall. Luajit project. http://www.luajit.org, 2016. [Online, accessed June 14, 2016].

M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Static Analysis*, pages 366–381. Springer, 1995.

M. Sperber, R. K. Dybvig, and M. Flatt. *Revised [6] report on the algorithmic language Scheme*, volume 19. Cambridge University Press, 2010.

R. Stallman, R. Pesch, S. Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002.

A. Wingo. value representation in javascript implementations. https://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations, 2011. [Online, accessed June 14, 2016].