An Interpreter for Server-Side Hop

Bernard Paul Serpette Manuel Serrano
Inria Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex – France
Bernard.Serpette@inria.fr Manuel.Serrano@inria.fr

Abstract

HOP is a Scheme-based multi-tier programming language for the Web. The client-side of a program is compiled to JavaScript, while the server-side is executed by a mix of natively compiled code and interpreted code. At the time where HOP programs were basic scripts, the performance of the server-side interpreter was not a concern; an inefficient interpreter was acceptable. As HOP expanded, HOP programs got larger and more complex. A more efficient interpreter was necessary. This new interpreter is described in this paper. It is compact, its whole implementation counting no more than 2.5 KLOC. It is more than twice faster than the old interpreter and consumes less than a third of its memory. Although it cannot compete with static or JIT native compilers, our experimental results show that it is amongst the fastest interpreters for dynamic languages.

1. Introduction

HOP is a Scheme based multi-tier programming language for the Web. We have implemented a new interpreter (Hopε0) for executing server-side parts of HOP programs1. By adapting techniques developed in other compiling contexts, we have accelerated the former Hopε0 interpreter by a factor of more than two, making Hopε1 one of the fastest interpreters we have measured.

Hopε0 accepted as input a tree structure close to the input source, which was obtained by parsing the source code, macro-expanding it, and resolving local variable references. These were interpreted as indexes into a heap-allocated lexical environment. In Hopε1, we significantly improve the interpreter efficiency by allocating local variables in a stack, as done in traditional compilation technique.

The Hopε1 interpreter is written in Hop and it is compiled to native code via C code generation. C is not properly tail recursive, i.e., calling a C function in a tail position allocates a stack frame. Hence, lacking special treatment, Hopε1 is not tail-recursive either. This is a critical problem because loops are traditionally implemented in Hop as tail recursive functions. In order to fix that problem, Hopε1 relies on a trampoline machinery2. A contribution of this paper is to show that a trampoline solves the tail recursion problem for interpreters such as Hopε1 without degrading performance.

There are various sorts of interpreters. Some evaluate the source code directly. Others construct a memory representation of the programs and apply simple transformations; Hopε0 falls into this category. Some interpreters pre-compile the program into an abstract structure that is interpreted by a dedicated virtual machine. Hopε1 leans toward a pre-compilation schema by deploying simple optimizations not implemented in Hopε0. A last contribution of this paper is to precisely evaluate the impacts of classical compile-time optimizations when applied to an interpreter.

The paper is organized as follows. Section 2 briefly introduces HOP and the context in which its interpreter is used. Section 3 presents the Hopε1 implementation techniques. Section 4 highlights some specific issues that impact the performance. Section 5 presents experimental performance results, comparing the new Hopε1 interpreter to interpreters for Scheme and other dynamic languages.

2. Background

Unlike traditional solutions for developing web applications, HOP [19] is a multi-tier programming language that makes it possible to write entire applications using a single formalism. With HOP, a web application is no longer a set of more or less loosely related components, such as PHP scripts on the server side and HTML+JavaScript on the client side. It is a single program written in a single programming language. HOP is largely inspired by the Scheme RSRS programming language [12], which it extends in several directions. It supports many modern features such as modules, objects, exceptions, and threads, coupled with a vast set of libraries. It also supports web-dedicated features: HTML elements are first class values;

1Work partially supported by the French ANR agency, grant ANR-09-EMER-009-01.

2See http://en.wikipedia.org/wiki/Trampoline_(computing) for a definition of the term trampoline and how it has been used in Lisp-like languages.
client-side programs are first class values of the server-side pro-
grams [3, 20]; and bidirectional communication between servers
and clients is supported.

A HOP program is organized in modules which describe the
server and client computations. The runtime environment of a HOP
application is thus twofold. The program is first installed on the
server. Then, the clients, i.e., the web browsers, receive their parts
of the application and communicate with their server. Figure 1
illustrates this multi-tier architecture.

![Figure 1. The Hop Architecture](image)

The HOP web server is bootstrapped. It is itself written in
HOP [17] and compiled by HOPC, a native compiler built on top
the Bigloo optimizing Scheme-to-C compiler [18]. The server
embeds the HOP-to-JavaScript compiler [14] which compiles
the client-side on-the-fly, and an interpreter which handles non-
natively compiled code on the server. The server-side parts of an
application are thus composed of a blend of natively compiled code
and interpreted code.

Resorting on interpreted code is optional as server-side parts
can be statically compiled if needed. For CPU-demanding applica-
tions, this approach is recommended. However, it comes with an
extra complexity because it requires the programmers to generate
dynamic libraries for each HOP-supported architecture. For desk-
tops that are equipped with full-fledged development kits, this is
probably not too heavy a burden. On constrained platforms such
as SmartPhones and single board computers that generally lack a
on-device C toolchain, heavyweight cross-compilation techniques
must be used.

At first, HOP applications were not CPU nor memory demand-
ing. The interpreter was mostly used for expanding macros con-
tained in the program source code and for running some hundred-
lines long scripts on the server side. The HOP\textsuperscript{E}\textsubscript{0} server-side inter-
pretation was accomplished by an hybrid engine that used tree
structures and a mix of native stack and separate heap-allocated
lexical environment. This interpreter offered acceptable CPU per-
formance, but, when HOP applications became larger and more com-
plex, its performance became unacceptable for three reasons:

- the heap allocation of the lexical environment stressed the
garbage collector too much;
- closures captured their whole lexical environment, making the
memory footprint unnecessarily large;
- loops where implemented using an \textit{ad hoc} tail-recursive inter-
pretation technique that significantly enlarged the size of each
native stack frame for non tail-recursive calls. This problem was
exacerbated by the HOP implementation of concurrency, which
relies on the Posix Threads where each thread stack is small
since pre-allocated at creation time.

The design and implementation of HOP\textsubscript{E}\textsubscript{1} has been motivated
by the elimination of these HOP\textsubscript{E}\textsubscript{0} drawbacks.

3. Interpreter Architecture

The implementation of HOP\textsubscript{E}\textsubscript{1} has been guided by five main con-
straints:

1. HOP is composed of a full-fledged web server, an optimizing
native compiler, an optimizing JavaScript client-code compiler,
a server-side interpreter, and a vast set of libraries. All this con-
stitutes a large and complex system, whose size is close to 500
HOP KLOC. Because of limited manpower, we could not afford
creating a new complex module for the server-side interpreta-
tion. We had to keep HOP\textsubscript{E}\textsubscript{1} as simple as HOP\textsubscript{E}\textsubscript{0}. To satisfy this
constraint, HOP\textsubscript{E}\textsubscript{1} is meta-circular, i.e., entirely implemented
in HOP, and it does not resort on any JIT compilation technique
nor specific virtual machine.

2. The HOP server-side environment is multi-threaded [17]. The
server handles each request in a dedicated Posix thread. Since
Posix threads pre-allocate fixed-size stacks, the interpreter
should not consume large stack frames for its own exe-
cution.

3. The HOP server is executed on all sorts of platforms, ranging
from standard desktop computers to single board computers or
embedded devices, which are generally not equipped with a
lot of memory. Avoiding unnecessary memory allocation and
memory retention is particularly important on such platforms.

4. Loops in HOP are expanded into tail-recursive calls. If not
implemented in this way, they these will raise stack overflows.

5. The HOPC compiler is used to build the server-side runtime
environment that contains the server-side HOP libraries. Library
functions are extensively used by interpreted code. For our
benchmark suite, more than 75% of the dynamic function calls
executed by the interpreter concern compiled native functions\textsuperscript{3}. A fast connection between interpreted code and compiled code
is of premium importance for the overall performance.

The HOP\textsubscript{E}\textsubscript{0} interpreter was satisfying the last two requirements. The HOP\textsubscript{E}\textsubscript{1} satisfies them all. In the rest of this section we present
its general architecture.

3.1 An Academic Interpreter

Which implementation language should be used for the interpreter?
Low-level languages such as assembly or C look most suitable
because they offer fine-grain control over low-level operations on
registers, stack, signals, etc. However, since the interpreted code
interacts with compiled code, it might be sensible to implement
the interpreter in the language compiled by the native compiler
and to compile it. This is the option we have chosen for HOP. It is
validated by the overall performance reported in Section 5.

We gradually present the interpreter architecture, starting with
the simple \(\lambda\)-calculus interpreter presented in Figure 2.

In such a denotational semantics based academic interpreter, the
lexical environment \(\epsilon\) could be simply represented by a function
mapping variables to values. However we use a standard associa-
tion list to show explicitly that each abstraction invocation allocates
two memory cells (line 9).

\textsuperscript{3}This accounts for functions like \texttt{map} or \texttt{read}, but also operators like + or
car.
1: (define (eval exp e)
2:   (match-case exp
3:     (atom ?x)
4:       (let ((slot (assq x e)))
5:         (if slot
6:           (cdr slot)
7:           null))
8:     (?x ?a)
9:       (λ (a) (eval body (cons (cons var x) e))))
10:      (?f ?a)
11:        ((?f ?a) ?body)
12:        (eval f e) (eval a e))
13:      (else null))

Figure 2. An academic interpreter.

The evaluation of an abstraction (line 8) returns an abstraction of the meta-language (line 9), making it possible for the meta-language to directly invoke evaluated functions. Using a different data-type would slow down the calls to interpreted functions executed by compiled code. Such calls typically occur when the interpreter calls the map library function.

3.2 A More Realistic Interpreter

Our first interpreter does not use any pre-compilation stage to save execution time. In the spirit of partial evaluation techniques, some computations may obviously be executed once and for all before the actual evaluation stage. For instance, the pattern matching that discriminates expressions can be statically performed once and for all before their evaluation takes place. This requires splitting the interpreter in two phases. First, a pre-compilation phase through the source code extracts simple static information. Then, a run phase computes the actual values.

The first phase, named II and pictured in Figure 3, receives the same arguments as the previous eval function. It returns values of type compiledExpr, whose constructor is named Ω. Henceforth, we will note (λ (v ... ) expr) a compiledExpr, where each v is a variable to be be bound at run time and expr is the run time code to be executed. The second phase, named Ω, takes as first argument a compiledExpr.

1: (define (II exp e)
2:   (match-case exp
3:     (atom ?x)
4:       (let ((slot (assq x e)))
5:         (if slot
6:           (λ (e) (cdr (assq x e)))
7:           null))
8:     (?x ?a)
9:       (λ (a) (II body (cons (cons var x) e))))
10:      (?f ?a)
11:        ((?f ?a) ?body)
12:        (II body (II body (cons (cons var x) e)))
13:      (else null))

Figure 3. Pre-compilation phase.

II only manipulates the symbolic part of the environment. The compile-time and run-time lexical environments have the same shape. Therefore, II can pre-compute indexes to be used by itself and Ω. These indexes correspond to De Bruijn indexes [7]. Thus, we can change the implementation of environments to a more compact and efficient representation: the compile-time environment becomes a list of symbols and the run-time environment becomes a list of values. The benefits are twofold. First, the new environment representations save memory space. Second, at run time, lookups over variable names are replaced with indexed accesses to lists.

This second version of II is shown in Figure 4. It has been first proposed by Feeley and Lapalme [9].

1: (define (II exp e)
2:   (match-case exp
3:     (atom ?x)
4:       (let ((slot (assq x e)))
5:         (if slot
6:           (λ (e) (list-ref e 1))
7:           null))
8:     (?x ?a)
9:       (λ (a) (II body (cons var e)))
10:      (?f ?a)
11:        ((?f ?a) ?body)
12:        (II body (II body (cons var e)))
13:      (else null))

Figure 4. Splitting compile-time and run-time environments.

3.3 Tree Interpretation

The representation of compiledExpr deeply impacts the implementation of Ω. We have tried two options: a tree structure; and a functional structure. The former, used by HOPE₀, is described in this section.

A value, (λ (v ... ) expr), is represented by an index and a heap-allocated data structure containing the values of the free variables occurring in expr and pointers to its sub-expressions. The whole Ω function is built by merging all the Λ expressions into an indexed switch as for a byte-code interpreter. An excerpt of the implementation of II can be found in Figure 5. An excerpt of the runtime evaluator Ω is given in Figure 6. In both cases, the integer 9 is the index of the function application.

1: (define (II exp e)
2:   (match-case exp
3:     (atom ?x)
4:       (let ((slot (assq x e)))
5:         (if slot
6:           (λ (e) (vector-ref e 0))
7:           null))
8:     (?x ?a)
9:       (λ (a) (II body (II body (cons (cons var x) e)))
10:      (?f ?a)
11:        ((?f ?a) ?body)
12:        (II body (II body (cons var e)))
13:      (else null))

Figure 5. Tree structure for function applications.

The meta-language of the interpreter, i.e., HOP itself, is simply tail recursive. That is, only syntactic self tail recursive calls are implemented without stack allocation. All other calls allocate stack frames whatever their context is. Thus, the recursive call for evaluating the body of a function allocates a stack frame, although it is in tail position. To get rid of this allocation, it is enough for the interpreter to recognize that the invoked function is interpreted, and thus to inline the call. The result of this transformation is given in Figure 6.

1: (define (Ω bc e)
2:   (case (vector-ref bc 0)
3:     (9)
4:       (let ((f (vector-ref bc 1))
5:          (a (vector-ref bc 2)))
6:         (λ (e) (II (II a e))))
7:       (else null))

Figure 6. Tree execution for function applications.
7. It merely puts the recursive call to $\Omega$ in a simple tail position that is immediately compiled by HOPC as a simple goto.

1: (define (\$bc $c\$))
2: (case (vector-ref bc 0) ...
11: (g9)
12: (let ((f ($\Omega$ (vector-ref bc 1) $c\$)))
13: (a ($\Omega$ (vector-ref bc 2) $c\$)))
14: (if (not (eval-procedure? f))
15: (f a)
16: (let ( ((body (eval-procedure-body f)))
17: (env (eval-procedure-env f)))
18: ($\Omega$ body (cons a env))))))
...

Figure 7. Properly tail-recursive interpreter.

The library predicate eval-procedure? returns true if and only if its argument is a procedure created by the interpreter. This predicate relies on a special HOPC feature that allows attributes to be associated with closures.

This version of the interpreter corresponds to HOPE, except for some optimizations that are not shown here. It satisfies the criteria 1, 4, and 5 of the introduction of Section 3, but it fails criteria 2 and 3. First, some programs have an unnecessarily large memory footprint because closures capture the whole lexical environment instead of a subset containing only their free variables. Second, the interpreter is properly tail-recursive but each regular function call allocates a very large stack frame. This is due to an artifact of the HOPC native compiler that we explain now.

The function $\Omega$ is mainly composed of a large switch, of which each branch corresponds to the execution of one type of expression. Some branches contain simple expressions such as literals or variable references. Other branches require complex computations. For example, for the sake of performance, the let forms are not macro-expanded in the branch that handles them, but are represented by an ad-hoc expression. Such a branch requires the introduction a lot of temporary variables. These variables will end up being allocated in the stack frame for $\Omega$ whatever branch is selected at runtime. Obviously, this wastes a lot of stack space. Unfortunately, there is no easy way out. First, because our tail recursion technique requires $\Omega$ to be self recursive. Second, because HOPC actually compiles to C and relies on regular C compilers for producing binary code. Actual stack frame allocation is based on C compilers we do not control. We have explored a different strategy that uses an additional stack. It will be presented in Section 3.5.

3.4 Functional Interpretation

The tree structure presented in the previous section can be considered as an ad hoc representation for closures. Each expression contains a code pointer denoted by an index and some closed values. The $\lambda$ constructor acts as an abstraction and can be replaced by a true closure. This transformation is shown in Figure 8.

With this representation, the runtime interpreter is a mere function application, as shown in Figure 9.

The new interpreter solves the large stack frames problem that plagues the tree interpreter, but it does not support proper tail-recursive calls. To avoid allocating stack frames, we will modify $\Omega$ to use a trampoline [21]. This transformation will be presented in Section 4.2.

---

3.5 A Second Stack

In all the interpreters presented so far, closures capture the entire lexical environment represented by a list (see Figure 4, line 12). The next step consists in replacing the heap allocation of the lexical environment with a stack allocation. After this modification, the interpreter will use two stacks: the implicit native stack used each time $\Omega$ is called and a new explicit interpreter stack used to allocate frames of interpreted functions.

The new version of the interpreter uses a global stack accessible by two library functions: stack-ref and stack-push!. The reference to a variable is thus an indexed read from the stack as shown in Figure 10.

1: (define ($\Omega$ exp $c\$))
2: (match-case exp ...
11: ((?a ...
12: (let ((f ($\Omega$ $c\$)))
13: (a ($\Omega$ $c\$)))
14: ((f $c\$) ($\Omega$ a $c\$))))
...

Figure 8. Functional representation for application.

1: (define ($\Omega$ cexp $c\$))
2: (cexp $c\$)

Figure 9. Functional execution for application.

Closures now need an additional heap-allocated structure to store their free variables. This is shown in line 13 in Figure 11. When the function is invoked, before its body is evaluated, an interpreted stack frame is allocated (see line 15 and line 16 in Figure 11).

... 8: ((?x ?y) ?body)
9: (let* ((vars (free-vars body $c\$))
10: (is (map (\?x \(index x) $c\$) vars))
11: (body ($\Omega$ body (cons a vars)))
12: ($\lambda$)
13: (let ((env (map (\?x \(index x) (stack-ref i)) is))
14: ($\lambda$ y)
15: (for-each stack-push! env)
16: (stack-push! x)
17: ...

Figure 10. Using an interpreter stack.

In the previous version of $\Pi$ (Figure 8), the function calls were fast but closures did capture their entire lexical stack, which made accessing a variable was linear in its nesting level. With the new
version (Figure 11), only the free variables are stored in the heap-allocated structure that is pushed on the stack each time the function is invoked. Accessing a variable becomes a constant-time operation. Whether one version of the interpreter is faster than the other depends on the nature of the source language to be interpreted. For a pure λ-calculus where each term has many free variables, the first strategy will probably be more efficient. The latter strategy is significantly more efficient for languages such as Scheme that propose a “let” form based on stack frames instead of heap-allocated environments. This is demonstrated by the experimental report presented in Section 5.

Several optimizations presented in Section 4 need a fine-grain control over the function call protocol. We decompose each evaluated function into two sub-functions: external and runner. The former is used by the compiler to invoke interpreted functions. The latter is used by the interpreter itself. The protocol to call external belongs to the compiler. The protocol to call runner uses the interpreter stack. Figure 12 shows the new version of the interpreter with these two functions.

```
1: ... 
8: ((?f ?a) ... 
9: (match-case exp
10: 3: ((?f (?var ?val)) ?body)
11: 4: (let ((?val ?f-body)) ?body)
12: 5: (let ((?var val)) ?body)
13: 6: (let ((?var val)) ?body)
14: 7: (let ((?var val)) ?body)
15: 8: (let ((?var val)) ?body)
16: 9: (let ((?var val)) ?body)
17: 10: (let ((?var val)) ?body)
18: 11: (let ((?var val)) ?body)
```

Figure 12. Re-arranging the closure environments with stacks to prepare future optimizations.

4. Optimizations

In this section, we present the various details of the implementation of HoPE and the main optimizations we have deployed.

4.1 Stack pointer vs Frame pointer

Stacks are generally accessed via a stack pointer, which is an index in a vector of values that points to the top of the stack. The distance between the stack pointer and a variable inside the stack is precisely the value of the variable’s De Bruijn index. For example, in the expression (λ (x) ... (let ((y ...))) ... x)) the De Bruijn index of x is 1 inside the let, which means that its value is stored one memory cell above the stack pointer.

Alternatively, a frame pointer can be preferred to a stack pointer. The frame pointer is constant during a function invocation, pointing to the first value pushed on the stack by the function call. Using a frame pointer is more efficient than using a stack pointer because it eliminates the need for incrementing and restoring the pointer each time a let form is evaluated (see Figure 13).

With a frame pointer, the distance between that pointer and the cell reserved for a variable value is known at compile-time. However it must be transmitted to the runtime interpreter.

4.2 Trampoline

The version of the interpreter presented in Figure 12 does not support proper tail recursion because the evaluation of the body of a function is embedded inside a λ-abstraction. To solve this problem, we have setup a trampoline [21] in HoPE. When the interpreter calls another interpreted function in a tail position it returns it to the current trampoline instead of calling the runner. A trampoline is installed when the interpreter calls another interpreted function in a non-tail position and when a compiled code calls an interpreted function. A trampoline invokes the function value it intercepts if and only if it corresponds to a runner. Otherwise, it returns it by to its caller. Trampolines recognize runners by checking a dedicated closure attribute, a HoPE-supported feature we briefly presented in Section 3.3. The code snippet in Figure 15 shows the trampolined version of the function call.

```
1: (define (II exp ?f) tail?)
2: (match-case exp
3: 3: ((??f ?a)
4: 4: (let ((f (II f II #f)) (a (II a II #f)))
5: 5: (λ ()
6: 6: (let ((f (II f)) (a (II a)))
7: 7: (if (not (eval-procedure? f))
8: 8: (f a)
9: 9: (let ((runner (eval-procedure-run f)))
10: 10: (let loop ((tmp (runner)))
11: 11: (if tail?
12: 12: runner
13: 13: (let loop ((tmp (runner)))
14: 14: (if (runner? tmp)
15: 15: (loop (tmp))
16: 16: (loop (tmp)))
17: 17: ... 
```

Figure 13. let evaluation with a stack pointer

```
1: (define (II exp ?f)
2: (match-case exp
3: 3: ((??f ?a)
4: 4: (let ((f (II f II #f)) (a (II a II #f)))
5: 5: (let ((?var val)) ?body)
6: 6: (let ((?var val)) ?body)
7: 7: (let ((?var val)) ?body)
8: 8: (let ((?var val)) ?body)
9: 9: (let ((?var val)) ?body)
10: 10: (let ((?var val)) ?body)
11: 11: (let ((?var val)) ?body)
```

Figure 14. let evaluation with a frame pointer

Figure 15. Properly tail-recursive interpreter with explicit trampolining.
specialized versions of runtime function calls: one for tail recursive calls and one for the other calls.

---

**Figure 16.** Impact of the trampoline on the Bglstone suite. Each dot represents the ratio of execution time with trampoline and execution time without trampoline, for one particular tested benchmark. For instance, the sieve benchmark runs 5% faster with trampoline than without, while the mbrot test runs about 3% slower with a trampoline.

---

Figure 16 shows the impact of trampolines on CPU performance. Each black dot represents one test of the Bglstone benchmark suite that contains 19 tests exercising different features of the Hop3. The dots are positioned on a single axis that represents the time ratio of the trampolined vs. non-trampolined versions. For instance, a dot located at position 0.90 means that for the associated test the execution with trampoline is faster and takes 90% of the time of the non-trampolined version. Surprisingly, there is a small speedup for most tests when trampolining is used. We suspect that this is due to a better cache locality of stack. One should note ratios less than 95% are insignificant because they are out of the range of uncertainty caused by memory cache and branch prediction effects.

### 4.3 Loops

In Hop, loops are implemented by local recursive functions, generally introduced by the general “letrec” special form, as in Figure 17. In this section, we show that although the trampoline presented in Section 4.2 avoids allocating stack for loops a special treatment of loops generally speeds up the execution.

---

**Figure 17.** Co-tail recursive functions forming a loop.

---

When all the variables defined in a “letrec” are bound to functions and they are exclusively used in tail-recursive calls and not closed under a lambda, the “letrec” is treated as a loop by the interpreter. It is modified to detect such loops and to use an ad hoc interpretation schema. It saves the symbolic environment at the entry point of the loop. This environment is a compile-time data structure that contains the names of the bound variables for each program point. In our example, when the loop is entered, the symbolic stack contains the identifier n. In the function odd?, it contains n and y. For each bound variable, the interpreter creates a function similar to the runner function presented in Section 4.2. This is a special case of a more general optimization found in several optimizing compilers, which consists in treating specially the variables bound to functions in a letrec. A complete study of this popular transformation can be found in [22].

When a tail call to a loop is encountered, the interpreter statically knows how many values must be dropped from the stack to adjust the stack at loop entry. For example, for the (even? tmp) call, the symbolic stack is (n tmp) and one value must be dropped. Arguments of the call are evaluated and saved in the stack. Finally, the runner associated with the local function is returned, the body of the “letrec” being evaluated as a trampoline.

---

Of the 391 “letrec” used in all the Bglstone suite, 144 are used to implement loops. The Figure 18 shows the impact of the loop detection on execution times. The optimization reduces execution times because it avoids allocating closures and references for captured variables.

---

**Figure 18.** Impact of the loop optimization on the Bglstone suite.

### 4.4 Inlining

Inlining is a well known technique that replaces a function call with a specialized version of the function body. Inlining is particularly important for functional languages that extensively use small functions. Hence, we have added inlining to HopE, keeping it simple enough to avoid significantly increasing the compilation time. We merely inline the body of a pre-defined set of frequent functions such as the fixnum arithmetic operators or the usual list accessors. Implementing this optimization consists in extending the Π function with some extra cases as shown in Figure 19.

---

**Figure 19.** Evaluation for some predefined functions.

---

Figure 20 presents the impact of the inlining optimization on the Bglstone benchmarks suite. It has a dramatic impact. For instance, it almost divides by 2 the execution time of tests such as fib.

---

**Figure 20.** Impact of the inlining optimization on the Bglstone suite.

### 4.5 N-ary functions

For the sake of simplicity, the interpreters presented in Section 3 assumed functions with exactly one argument. Extension to n-argument calls is straightforward. While the protocol used by the interpreter to call an interpreted function is independent of the number of actual values (see the runner function presented in Section 4.2), the protocol used for invoking a compiled function from the interpreter depends on the number of arguments. This protocol consists in apply-ing compiled functions to a list of packed arguments, allocating lists from which the compiled code extracts the actual values. This method is presented in Figure 21.

To avoid packing the arguments in lists, the interpreter treats specially function calls with 0 up to N arguments. What should be the value of N for improving the performance? We have conducted another experiment reported in Figure 22. We have measured the execution times of each benchmark for N in the range [1...6],

---

3 ftp://ftp-sop.inria.fr/index/ftp/Bigloo/apps/bglstone.tar.gz
1: (define (Π exp ϵ Π))
2: (match-case exp
3: ((?f . ?args)
4:  (let ((f (Π f ϵ Π))
5:          (args (map (λ (a) (Π a ϵ Π)) args)))
6:           (Λ ()
7:             (let ((f (Ω f)))
8:               (if (not (eval-procedure? f))
9:                 (apply f (map Ω args))
10:                  (begin
11:                    (eval-and-push args)
12:                    (call-interpreterd-function f))))))
13: ...)

Figure 21. Call with undefined number of arguments.

comparing it to the execution time of the same benchmark with \(N = 4\). This experiment shows that a special handling of 1 to 4 arguments is beneficial. We excluded functions with 0 argument because these functions are always compiled specially.

As shown in Figure 10, when the interpreter pre-compiles a user function, it generates two functions: the runner and external. The latter is used when the compiled code calls back the interpreter, for instance, when a compiled library function such as \texttt{map} calls its first argument. We have measured the impact of compiling the external function. It is reported in Figure 23, which shows that the protocol used to invoke an interpreted function from compiled code does not impact the overall performance. So, the interpreter always uses a single external function for all cases.

4.6 Debugging

For the sake of debugging, the interpreter computes a symbolic stack of called functions at any program point. Figure 25 shows the overhead imposed by computing that information on the benchmark tests. The maximal overhead is about 40% with a mean smaller than 15%. We have considered this overhead as affordable. Thus we keep debugging enabled. For a similar reason, the interpreted code always checks the dynamic type of values before accessing any data structure.

4.7 Extensible Stacks

The interpreter stack is allocated in the heap and represented as a list of HOP vectors. When an interpreted function is called, the free space remaining on the stack is checked. When it is too small, a new vector is allocated and added to the list. The stack is then accessed using unsafe library functions that do not check the vectors bounds. Figure 26 presents the overhead imposed by checking the stack size when functions are entered.

We have measured the impact of the vector sizes on the performance of the interpreter. This is presented in Figure 27. It shows that the maximal performance is reached from 8KB vectors, which is large enough to run all the benchmarks without extending the stack.
5. Evaluation

In this section, we compare the performance of Hop\textsuperscript{E}\textsubscript{1} with other implementations of dynamic languages. First, we compare it to various implementations of Scheme, the language on top of which Hop is built. Then, we compare it with a broader spectrum of languages.

The experimental values are collected on two platforms: an Intel x86/32 Xeon W3570 3.2Ghz running Linux-2.6.39 and a Samsung S3C2440 ARM9 400Mhz running Linux-2.6.32. To measure the execution times, each program is executed three times and the minimal system+user sum is collected. When possible, benchmarks have been configured for the execution times of the new interpreter to be approximately 10 seconds on the x86 platform. Using a long enough execution minimizes the impact of the boot-time of each system. This corresponds to the way Hop\textsuperscript{E}\textsubscript{1} is used in Hop because there is no real boot-time where the server is already fully initialized and the program loaded and compiled. But systems which optimize start times versus run times may be penalized in this measurement.

5.1 Hop vs Scheme Implementations

To estimate the efficiency of the new Hop\textsuperscript{E}\textsubscript{1} interpreter, we have first compared it to other implementations of Scheme. The implementations we have used are the following.

- **Bigloo 3.6b and Racket 5.1.1**: Bigloo is the native of Hop via C. Racket is a JIT compiler for Scheme. This comparison shows that interpreters cannot compete with true compilers and that native compilation is unavoidable when performance is really needed.
- **Hop\textsuperscript{E}0 and Hop\textsuperscript{E}\textsubscript{1}** are the two Hop interpreters. Although an important speedup is unsurprising because Hop\textsuperscript{E}0 was not optimized for speed, this comparison is important because it let us measure to which extend our primary goal has been reached.
- **Gambit-C 4.6.1.** Gambit is a Scheme implementation that contains a C code compiler and a meta-circular interpreter which uses explicit closures as for Hop\textsuperscript{E}\textsubscript{1}. However, in Gambit-C, closures are encoded with Scheme vectors instead of Scheme procedures. Gambit-C is configured with --enable-single-host to improve its performance. We only measure the execution times of the Gambit-C interpreter that is started with a 4MB heap.
- **Guile 2.0.2** is the GNU extension language. It is a popular Scheme implementation with a byte-code interpreter written in C.
- **STklos 1.01+** is a byte code interpreter implemented in C.

Comparing various implementations is complex because it is difficult to configure and use them similarly enough. Some systems can only partially disable the debugging support, some support specialized arithmetic while others don’t, etc. Therefore, the feature difference between two systems may impact their overall performance more than the quality of their implementation. For instance, Hop does not support Scheme’s ecall/cc. Lacking this feature enables optimizations that are forbidden for pure Scheme implementations. So, only the comparison of the execution times of Hop\textsuperscript{E}0 and Hop\textsuperscript{E}\textsubscript{1} is totally meaningful: their execution times are measured on two systems differing only by the interpreters. As far as the comparison with other systems is concerned, we think that only big enough ratios (greater than 50%) are meaningful.

The Scheme benchmarks use specific arithmetic operators when available (for instance, Hop uses +fx, Gambit ##fxsum+, and STklos and Racket fx+). All the Scheme implementations benefit from this specialization except Guile, which only supports generic operators. Operators are inlined by systems that support modules, e.g., Bigloo and Racket, because they use module boundaries to detect that the operators are constant functions. Execution is safe since the types are checked at runtime. All systems but Bigloo, Guile, and probably Racket offer some debugging facilities (see Section 4.6).

The benchmarks timings of the Bglstone suite are reported in Figures 29 and 30. The first observation is that no interpreter can compete with compiled code. Bigloo is the faster compiler tested here. It is one or two orders of magnitude faster than Guile and Hop\textsuperscript{E}\textsubscript{1}, the faster interpreters. Guile is slightly faster that Hop\textsuperscript{E}\textsubscript{1} but their execution environments are not totally equivalent. In particular, Guile does not support specific arithmetic operations; adding such functions could significantly increase its performance. Another observation is that Hop\textsuperscript{E}\textsubscript{1} is more than twice faster than Hop\textsuperscript{E}0. Hop\textsuperscript{E}\textsubscript{1} satisfies its general goal to eliminate the drawbacks of Hop\textsuperscript{E}0 without jeopardizing the overall performance. The last observation is that Guile and Hop\textsuperscript{E}\textsubscript{1} are roughly two or three times faster than Gambit and STklos. This indicates that the implementa-
We have compared the memory allocations of \( \text{Hop}^\mathcal{E}_1 \) and \( \text{Hop}^\mathcal{E}_0 \). The result is presented in Figure 31. Note that for this experiment we have executed the tests with smaller input values for the tests. \( \text{Hop}^\mathcal{E}_1 \) reduces significantly the allocated memory, benefiting from its stack structure (see Section 3.5). Greater amounts of allocated memory for \( \text{Hop}^\mathcal{E}_0 \) do not necessarily imply that it has a larger memory footprint, because most of the allocated stack frames have a very short life time and are reclaimed at each collection. This is why the ratios between the memory allocated and number of executed collections differ. Surprisingly, for some tests, the CPU time ratio and the allocated memory ratio seem unrelated. For instance \( \text{Hop}^\mathcal{E}_0 \) allocates 350 times more memory than \( \text{Hop}^\mathcal{E}_1 \) for \texttt{fib} and runs 478 times more collections, while its execution is only 1.6 times slower. We have no explanation to propose to that surprising result. On the other hand, both interpreters consumes the same amount of memory for \texttt{fft} but \( \text{Hop}^\mathcal{E}_1 \) executes it twice faster. Other tests such as \texttt{puzzle} or \texttt{leval} are more conform to the intuition that the execution time is strongly related to the amount of allocated memory.

### 5.2 Hop vs Dynamic Languages

In this section, we compare the performances of the two \( \text{Hop} \) interpreters to the performances of other dynamic languages. The methodology differs from the previous section. Each system comes with its own implementation of each benchmark, reflecting the idiosyncrasies of both the language and the evaluation engine. This section is not a fine grain comparison. It merely gives a general idea of the maximal performance each system may deliver. A difference of 20% or 30% is not significant here, only the orders of magnitude mattering. If one system is 10 times slower than the others for all the benchmarks, it tells us something about the complexity limitation of the problems that its language can address. Jumping to conclusions when the differences are smaller would probably be misleading.

For that experiment, we consider seven programming languages: JavaScript v8 3.2.10, Lua 5.1.4, Perl 5.12.3, PHP 5.3.6, Python 3.2, Ruby 1.9.2, and \texttt{Hop}. We have used 5 test programs. First, \texttt{fib} because it is simple enough to be implemented as is in all languages and because it is a good test for measuring the raw performance of function invocation and the exact arithmetic. Then, we have used four tests adapted from the Computer Language Benchmarks Game, aka Shoothout.

The Computer Language Benchmarks Game aims at comparing the best possible performance delivered by of all sort of implement-
<table>
<thead>
<tr>
<th>Bench</th>
<th>HopE₀</th>
<th>HopE₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>alambench</td>
<td>13MB - 7</td>
<td>25MB - 28</td>
</tr>
<tr>
<td>bague</td>
<td>3MB - 1</td>
<td>71MB - 3</td>
</tr>
<tr>
<td>beval</td>
<td>7MB - 3</td>
<td>144MB - 69</td>
</tr>
<tr>
<td>boyer</td>
<td>5MB - 2</td>
<td>57MB - 30</td>
</tr>
<tr>
<td>conform</td>
<td>12MB - 9</td>
<td>46MB - 22</td>
</tr>
<tr>
<td>earley</td>
<td>14MB - 7</td>
<td>135MB - 87</td>
</tr>
<tr>
<td>fit</td>
<td>62MB - 76</td>
<td>84MB - 80</td>
</tr>
<tr>
<td>fib</td>
<td>3MB - 1</td>
<td>1050MB - 478</td>
</tr>
<tr>
<td>leval</td>
<td>683MB - 34</td>
<td>756MB - 347</td>
</tr>
<tr>
<td>maze</td>
<td>10MB - 3</td>
<td>92MB - 19</td>
</tr>
<tr>
<td>mbrot</td>
<td>51MB - 59</td>
<td>116MB - 90</td>
</tr>
<tr>
<td>molec</td>
<td>46MB - 23</td>
<td>130MB - 70</td>
</tr>
<tr>
<td>peval</td>
<td>32MB - 27</td>
<td>46MB - 23</td>
</tr>
<tr>
<td>puzzle</td>
<td>3MB - 1</td>
<td>263MB - 124</td>
</tr>
<tr>
<td>quest</td>
<td>5MB - 2</td>
<td>128MB - 77</td>
</tr>
<tr>
<td>queens</td>
<td>14MB - 55</td>
<td>352MB - 97</td>
</tr>
<tr>
<td>serve</td>
<td>7MB - 3</td>
<td>27MB - 13</td>
</tr>
<tr>
<td>traverse</td>
<td>8MB - 4</td>
<td>316MB - 131</td>
</tr>
</tbody>
</table>

Figure 31. Compared memory allocations for Bglstone. For each benchmark, the amount of allocated memory is reported in megabytes and followed by the number of collections.

As a conclusion of the Bglstone experiment, interpreted code cannot compete with compiled code. Bigloo and JavaScript V8 are one or two orders of magnitude faster than interpreters. Within interpreters, HopE₁ delivers good performances for fib, btrees, and norm, but poor performances for fasta and mbrot. These two tests are floating-point intensive. For instance, the whole execution time of mbrot is spent executing the following sequence that is constantly repeated:

```scheme
(let loop ((i 0))
  (let ((Zi (+2.0 (*Zr Zi))))
      (set! Zi (+ (*Zr *Ti) Ci)))
  (let ((Tr (-(*1 Tr Ti) Cr))
      (set! Tr (+2.0 *Zr))
      (set! Ti (*1 Zi Zi)))
      (set! k (+fx k 5)))
  (when (< (*1 (*1 Tr Ti) 4.0)) (loop (+fx i 1))))
```

Although the floating optimization presented Section 4.8 unboxes numbers for subexpressions, the execution time of this test is still largely dominated by the time required to allocate and collect the numbers stored in the Zi, Zr, Tr, and Ti variables. The whole execution allocates about 600MB of floating numbers and runs 76 collections to reclaim them. Improving the performance of HopE₁ for this kind of test will require a new optimization for numbers. The compiled code does not suffer the same problem because its static analysis eliminates all boxing operations at compile time.

### 6. Related Work

Implementing efficient interpreters seems to be almost as old as computer science. For instance a paper by Klint from 1981 [13] contains a comparison of compilation and interpretation. A long debate opposes byte-code switch to threaded code [1, 5, 8, 15]. The function operators of HopE₁ might be considered as a variant of threaded code. Our experiment is yet another example where threaded code delivers faster code then byte-code switches. Another paper by Brunthaler [4] studies the impact of low level code representation and presents some general optimization such as caching interpreted local variables that could be used in HopE₁.

As for Javascript, the Lua programming language is designed for simplicity. It only supports floating-point numbers. It emulates arrays with association tables and supports only few constructs with a limited syntax. Its implementation is based on a remarkably efficient and compact byte-code interpreter. The whole implementation fits in less than 17 KLOC of C code. Instead of using a stack as for the seminal Pascal’s P-machine [16], it uses a register-based architecture [11]. This architecture avoids many `pop` and `push` instructions that stack-based code needs to move values around the stack. Since it avoids boxing temporary results, it is particularly efficient for dealing with floating point numbers. This is reflected by the good performance of the Lua interpreter on the all floating point intensive Shootout tests fasta, mbrot, and snorm.

Python is a byte-code interpreter implemented in C. It uses threaded code instead of a byte-code switches; accordingly to a comment located inside the source code, this improves the performance by 15-20% [6]. Alternative systems for executing Python are under investigations. In particular, the PyPy experiment consists in implementing a meta-circular Python JIT. This work concludes that high-level languages are suitable for implementing dynamic languages [2]. We have reached the same conclusion.

Mainstream JavaScript implementations such as Firefox TraceMonkey [10] or Google Chrome V8 use JIT compilers to improve performance. As shown in this section, this technique significantly outperforms interpreters at the price of a higher complexity, but developing such compilers is much more complex than developing interpreters. Unsurprisingly, they are also less portable. This probably explains why some of them are only available on x86 architectures.

PHP has a threaded code interpreter for a dynamic typed virtual machine called Zend. The interpreter is written in C. PHP relies on reference-count garbage collection. Allocating the results of arithmetic operations is carefully avoided. For example, the virtual machine proposes 16 different operators for implementing multiplications.

### 7. Conclusion

We have presented HopE₁, the new HOP interpreter that works hand-in-hand with native compiled code to execute the server-side parts of HOP programs. This new interpreter is meta-circular and implemented in HOP. Its source code is no more than 2.5 KLOC. This compactness made it simple to develop and now makes it easy to maintain.

HopE₁ has fulfilled its primary goal, which was to eliminate the major weaknesses of the former HopE₀ interpreter. As demonstrated in Section 5, HopE₁ has significantly exceeded that goal. It is one of the fastest Scheme interpreters we have tested. HopE₁ is between two and three times faster than HopE₀. Although performance still lags far behind compiled code, such a speedup is important for HOP. A two-fold speedup greatly reduces electricity consumption and heat emission, which is critical for portable devices or for future fan-less computers. This is an important property for application domains such as multimedia, which are primary targets for HOP.

### 8. Acknowledgements

Many thanks to Ludovic Courtois, Eric Gallesio, Marc Feeley, and Gérard Berry for their helpful feedback on this work.
### References


