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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Language Classifications—applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—interpreters; I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems—evaluation strategies

General Terms Languages, Experimentation, Measurement, Performance

Keywords Functional languages, Scheme, Implementation, Interpreter

1. Introduction

HOP is a Scheme based multi-tier programming language for the Web. We have implemented a new interpreter (HOP$\mathcal{E}_1$) for executing server-side parts of HOP programs[1]. By adapting techniques developed in other compiling contexts , we have accelerated the former HOP$\mathcal{E}_0$ interpreter by a factor of more than two, making HOP$\mathcal{E}_1$ one of the fastest interpreters we have measured.

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

DLS'11, October 24, 2011, Portland, Oregon, USA.

HOP$\mathcal{E}_0$ 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 HOP$\mathcal{E}_1$, we significantly improve the interpreter efficiency by allocating local variables in a stack, as done in traditional compilation technique.

The HOP$\mathcal{E}_1$ 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, HOP$\mathcal{E}_1$ 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, HOP$\mathcal{E}_1$ relies on a trampoline machinery[2]. A contribution of this paper is to show that a trampoline solves the tail recursion problem for interpreters such as HOP$\mathcal{E}_1$ 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; HOP$\mathcal{E}_0$ falls into this category. Some interpreters pre-compile the program into an abstract structure that is interpreted by a dedicated virtual machine. HOP$\mathcal{E}_1$ leans toward a pre-compilation schema by deploying simple optimizations not implemented in HOP$\mathcal{E}_0$. 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 HOP$\mathcal{E}_1$ implementation techniques. Section **4** highlights some specific issues that impact the performance. Section **5** presents experimental performance results, comparing the new HOP$\mathcal{E}_1$ 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 R5RS 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;

[2] See `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 programs [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.

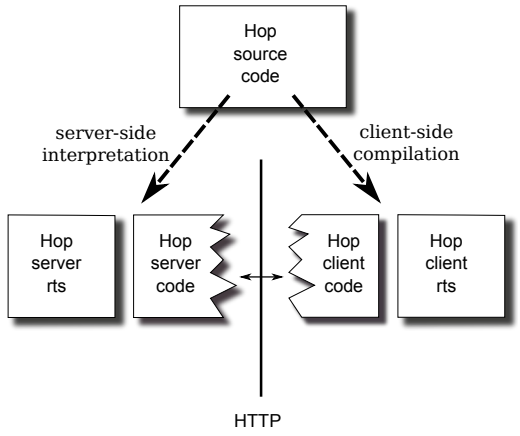

Figure 1. The Hop Architecture

The HOP web server is bootstrapped. It is itself written in HOP [17] and compiled by HOP$\mathcal{C}$, 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 applications, 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 desktops 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 demanding. The interpreter was mostly used for expansing macros contained in the program source code and for running some hundred-lines long scripts on the server side. The HOP$\mathcal{E}_0$ server-side interpretation 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 performance, but, when HOP applications became larger and more complex, 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 *ad hoc* tail-recursive interpretation 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$\mathcal{E}_1$ has been motivated by the elimination of these HOP$\mathcal{E}_0$ drawbacks.

3. Interpreter Architecture

The implementation of HOP$\mathcal{E}_1$ has been guided by five main constraints:

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 constitutes 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 interpretation. We had to keep HOP$\mathcal{E}_1$ as simple as HOP$\mathcal{E}_0$. To satisfy this constraint, HOP$\mathcal{E}_1$ 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 not consume large stack frames for its own execution.
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 HOP$\mathcal{C}$ 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[3]. A fast connection between interpreted code and compiled code is of premium importance for the overall performance.

The HOP$\mathcal{E}_0$ interpreter was satisfying the last two requirements. The HOP$\mathcal{E}_1$ 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 λ-calculus interpreter presented in Figure 2.

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

[3] This accounts for functions like `map` or `read`, but also operators like `+` or `car`.

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

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 Π and pictured in Figure 3, receives the same arguments as the previous *eval* function. It returns values of type *compiledExpr*, whose constructor is named Λ. 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 (Π exp ϵΠ)
2:   (match-case exp
3:     ((atom ?x)
4:      (let ((slot (assq x ϵΠ)))
5:        (if slot
6:           (Λ (ϵΩ) (cdr (assq x ϵΩ)))
7:           ⊥)))
8:     ((λ (?v) ?body)
9:      (let ((body (Π body (cons (cons v #f) ϵΠ))))
10:       (Λ (ϵΩ)
11:         (λ (x)
12:           (Ω body (cons (cons v x) ϵΩ))))))
13:    ((?f ?a)
14:     (let ((f (Π f ϵΠ)) (a (Π a ϵΠ)))
15:       (Λ (ϵΩ)
16:         ((Ω f ϵΩ) (Ω a ϵΩ)))))
17:    (else ⊥)))
```

Figure 3. Pre-compilation phase.

Π only manipulates the symbolic part of the environment. The compile-time and run-time lexical environments have the same shape. Therefore, Π 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 Π is shown in Figure 4. It has been first proposed by Feeley and Lapalme [9].

```
1: (define (Π exp ϵΠ)
2:    (match-case exp
3:       ((atom ?x)
4:        (let ((i (index x ϵΠ)) )
5:           (if i
6:               (Λ (ϵΩ) (list-ref ϵΩ i))
7:               ⊥ )))
8:       ((λ (?var) ?body)
9:        (let ((body (Π body (cons var ϵΠ))))
10:          (Λ (ϵΩ)
11:             (λ (x)
12:                (Ω body (cons x ϵΩ))))))
...
```

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

3.3 Tree Interpretation

The representation of *compiledExpr* deeply impacts the implementation of Λ. We have tried two options: a tree structure; and a functional structure. The former, used by HOP$\mathcal{E}_0$, 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 Π 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 (Π exp ϵΠ)
2:    (match-case exp
...
11:      ((?f ?a)
12:       (let ((f (Π f ϵΠ)) (a (Π a ϵΠ)))
13:          (vector 9 f a)))
...
```

Figure 5. Tree structure for function applications.

```
1: (define (Ω bc ϵΩ)
2:    (case (vector-ref bc 0)
...
11:      ((9)
12:       (let ((f (vector-ref bc 1))
13:             (a (vector-ref bc 2)))
14:          ((Ω f ϵΩ) (Ω a ϵΩ))
15: ...
```

Figure 6. Tree execution 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

7. It merely puts the recursive call to Ω in a simple tail position that is immediately compiled by HOP$\mathcal{C}$ as a simple `goto`.

```
 1: (define (Ω bc ϵΩ)
 2:    (case (vector-ref bc 0)
...
11:       ((9)
12:        (let ((f (Ω (vector-ref bc 1) ϵΩ))
13:              (a (Ω (vector-ref bc 2) ϵΩ)))
14:           (if (not (eval-procedure? f))
15:               (f a)
16:               (let ((body (eval-procedure-body f))
17:                     (env (eval-procedure-env f)))
18:                  (Ω 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 HOP$\mathcal{C}$ feature that allows attributes to be associated with closures.

This version of the interpreter corresponds to HOP$\mathcal{E}_0$, 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 HOP$\mathcal{C}$ native compiler that we explain now.

The function Ω 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 and *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 Ω 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 that all recursive calls to Ω are self recursive. Second, because HOP$\mathcal{C}$ actually compiles to C and relies on regular C compilers for producing binary code. Actual stack frame allocation depends on C compilers we do not control[4]. 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 Λ 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 Ω to use a trampoline [21]. This transformation will be presented in Section 4.2.

[4] This phenomenon has also been observed by the author of the LuaJIT compiler in a note available at `http://article.gmane.org/gmane.comp.lang.lua.general/75426`.

```
 1: (define (Π exp ϵΠ)
 2:    (match-case exp
...
11:       ((?f ?a)
12:        (let ((f (Π f ϵΠ)) (a (Π a ϵΠ)))
13:           (λ (ϵΩ)
14:             ((Ω f ϵΩ) (Ω a ϵΩ)))))
...
```

Figure 8. Functional representation for application.

```
1: (define (Ω cexp ϵΩ)
2:    (cexp ϵΩ))
```

Figure 9. Functional execution for application.

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 Ω 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 (Π exp ϵΠ)
2:    (match-case exp
3:       ((atom ?x)
4:        (let ((i (index x ϵΠ)))
5:           (if i
6:               (Λ () (stack-ref i))
7:               ⊥)))
...
```

Figure 10. Using an interpreter stack.

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: ((λ (?var) ?body)
 9:  (let* ((vars (free-vars body ϵΠ))
10:         (is (map (λ (x) (index x ϵΠ)) vars))
11:         (body (Π body (cons var vars))))
12:    (Λ ()
13:      (let ((env (map (λ (i) (stack-ref i)) is)))
14:        (λ (x)
15:          (for-each stack-push! env)
16:          (stack-push! x)
17:          ...evaluate body and restore stack...)))))
...
```

Figure 11. Closure environments with stacks.

In the previous version of Π (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: ((λ (?var) ?body)
9:  (let* ((vars (free-vars body ϵ∏))
10:        (is (map (λ (x) (index x ϵ∏)) vars))
11:        (body (Π body (cons var vars))))
12:    (Λ ()
13:      (let* ((env (map (λ (i) (stack-ref i)) is))
14:             (runner (λ ()
15:               ...extra bookkeeping...
16:               (for-each stack-push! env)
17:               ...evaluate body and restore stack...))
18:             (external (λ (x)
19:               (stack-push! x)
20:               (runner))))
21:        external)))
22: ...
```

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 HOP$\mathcal{E}_1$ 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 in 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 Brujin 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 in 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 HOP$\mathcal{E}_1$. 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 *runner*s by checking a dedicated closure attribute, a HOP$\mathcal{C}$-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 (Π exp ϵ∏)
2:   (match-case exp
3:     ((let ((?var ?val)) ?body)
4:      (let ((val (Π val ϵ∏))
5:            (body (Π body (cons var ϵ∏))))
6:        (Λ ()
7:           (let ((val (Ω val)) (oldsp sp))
8:              (set! sp (+ sp 1))
9:              (stack-set! sp val)
10:             (prog1
11:                (Ω body)
12:                (set! sp oldsp))))))
13: ...
```

Figure 13. let evaluation with a stack pointer

```
1: (define (Π exp ϵ∏)
2:   (match-case exp
3:     ((let ((?var ?val)) ?body)
4:      (let ((val (Π val ϵ∏))
5:            (body (Π body (cons var ϵ∏))))
6:        (let ((i (length ϵ∏)))
7:           (Λ ()
8:              (let ((val (Ω val)))
9:                 (stack-set! (+ fp i) val)
10:                (Ω body))))))
11: ...
```

Figure 14. let evaluation with a frame pointer

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

Figure 15. Properly tail-recursive interpreter with explicit trampolining.

Since the value of the argument `tail?` is known at compile time, the test used to either return a continuation value or install a new trampoline can be lifted out of the Λ form, which leads to two

specialized versions of runtime function calls: one for tail recursive calls and one for the other calls.

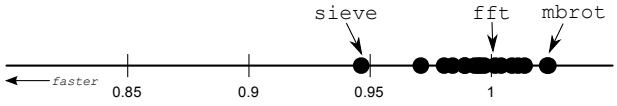

Figure 16. Impact of the trampoline on the Bglstone suite. Each dot represents the ratio of execution time with trampoline and execution time without trampoline, for one particular tested benchmark. For instance, the `sieve` benchmark runs 5% faster with trampoline than without, while the `mbrot` test 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 exercizing different features of the HOP[5]. 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.

```
1: (define (is-even? n)
2:   (letrec ((even? (λ (x)
3:                     (if (= x 0) #t (odd? (- x 1)))))
4:            (odd? (λ (y)
5:                     (if (= y 0) #f (even? (- y 1))))))
6:     (let ((tmp n))
7:        (even? tmp))))
```

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

When all the variables defined in a "letrec" are bound to functions and when they are exclusively used in tail-recursive calls and not closed under a lambda, the "letrec" is treated as a loop by the interpreter. Π is modified to detect such loops and to use an *ad hoc* interpretation schema. Π 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 evaluted 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.

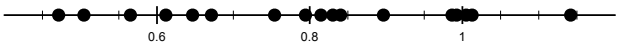

Figure 18. Impact of the loop optimization on the Bglstone suite.

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 HOP$\mathcal{E}_1$, 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.

```
1: (define (Π exp ϵΠ)
2:    (match-case exp
3:       ((+ ?x ?y)
4:        (let ((x (Π x ϵΠ)) (y (Π y ϵΠ)))
5:           (Λ ()
6:              (+ (Ω x) (Ω y)))))
7: ...
```

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`.

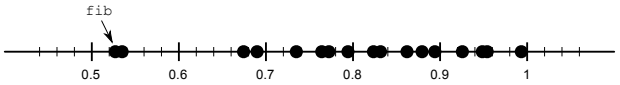

Figure 20. Impact of the inlining optimization on the Bglstone suite.

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 $[1...6]$,

[5] `ftp://ftp-sop.inria.fr/indes/fp/Bigloo/apps/bglstone.tar.gz`

```
1: (define (Π exp ϵΠ)
2:   (match-case exp
3:     ((?f . ?args)
4:      (let ((f (Π f ϵΠ))
5:            (args (map (λ (a) (Π a ϵΠ)) 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.

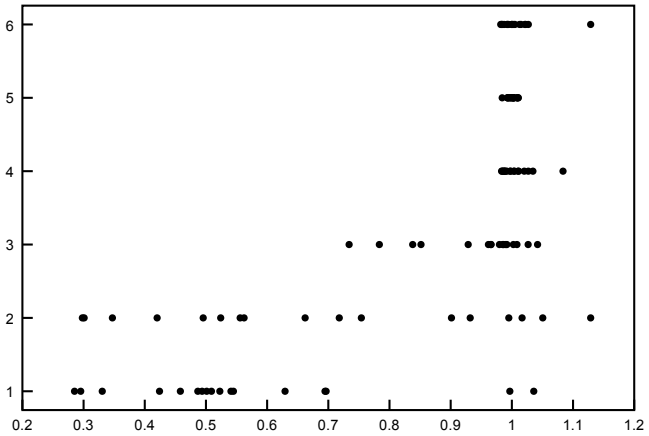

Figure 22. Impact of the specialized function call protocol. The vertical axis is the threshold from which the generic protocol is used. The horizontal axis is the ratios of the optimized execution times divided by the generic executions times.

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 `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

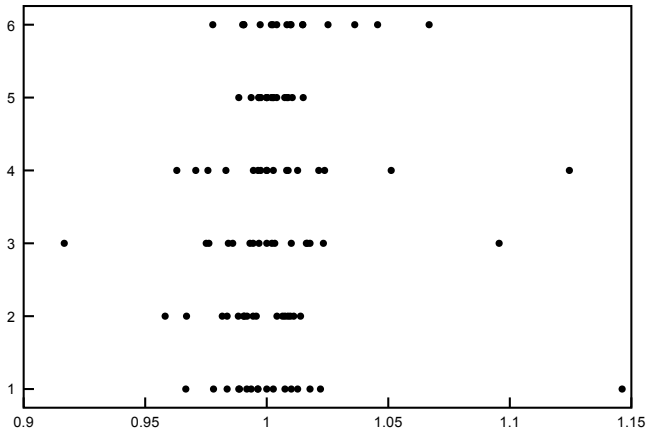

Figure 23. Impact of the compilation scheme used for the *external* function used the compiler calls back an interpreted function. The vertical axis represents the number of arguments. The horizontal axis represents the execution time ratios between special and generic compilation.

```
1: (match-case exp
2:  ((λ (?var . ?rest) ?body)
3:   (let (...)
4:    (Λ ()
5:      (let* ((env (map (λ (i) (stack-ref i)) is))
6:             (runner (λ ()
7:               ...extra bookkeeping...
8:               (for-each stack-push! env)
9:               ...evaluate body and restore stack...)))
10:       (let ((external (λ (x . l)
11:                (stack-push! x)
12:                (bind-frame rest l)
13:                (runner))))
14:         external)))
15: ...
```

Figure 24. Abstraction with undefined number of arguments.

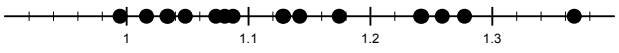

Figure 25. Impact of debugging information on the execution times for the Bglstone benchmark suite.

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.

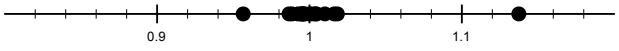

Figure 26. Impact of the stack bound checking on the performance.

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.

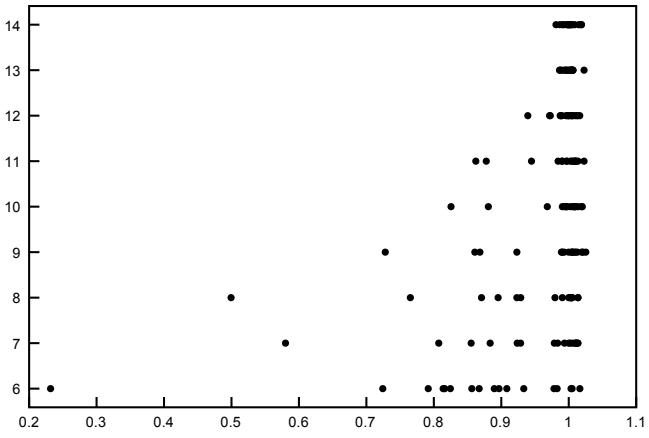

Figure 27. Impact of the stack chunks size on the overall performance. The vertical axis is stack of the chunks expressed in 2^x bytes. The horizontal axis is the ratio of the execution times divided by the execution time of the actual interpreter.

4.8 Arithmetic expressions

HOP uses 64 bit allocated floating point numbers. Therefore, floating point operators have to *unbox*, compute, and *box*. Boxing and unboxing dramatically slows down execution. So, we have added a simple optimization that consists in using a dedicated interpreter for floating point expressions. When the standard Π interpreter parses a floating point operator, it generates a specific function that invokes the dedicated interpreter. This interpreter benefits from the HOP$\mathcal{C}$ optimization that unboxes floating point numbers. Such a floating point optimization avoids allocating numbers for temporary results, but it still requires to allocate one fresh number per arithmetic expression for the result. Figure 28 shows the impact of this optimization on the Bglstone tests. The highest speedup is observed for the `nucleic` test, whose execution is about 30% faster when the optimization is enabled.

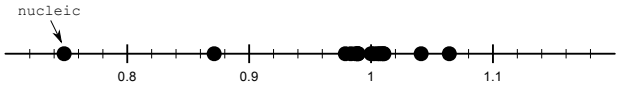

Figure 28. Impact of the arithmetic optimization.

5. Evaluation

In this section, we compare the performance of HOP$\mathcal{E}_1$ 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 approximatively 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$\mathcal{E}_1$ 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 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$\mathcal{E}_0$** and **HOP$\mathcal{E}_1$** are the two HOP interpreters. Although an important speedup is unsurprising because HOP$\mathcal{E}_0$ 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$\mathcal{E}_1$. 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 `call/cc`. Lacking this feature enables optimizations that are forbidden for pure Scheme implementations. So, only the comparison of the execution times of HOP$\mathcal{E}_0$ and HOP$\mathcal{E}_1$ 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 `##fixnum.+`, 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$\mathcal{E}_1$, the faster interpreters. Guile is slightly faster that HOP$\mathcal{E}_1$ 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$\mathcal{E}_1$ is more than twice faster than HOP$\mathcal{E}_0$. HOP$\mathcal{E}_1$ satisfies its general goal to eliminate the drawbacks of HOP$\mathcal{E}_0$ without jeopardizing the overall performance. The last observation is that Guile and HOP$\mathcal{E}_1$ are roughly two or three times faster than Gambit and STklos. This indicates that the implementa-

	Interpreters					*Compilers*	
Bench	**Hop$\mathcal{E}_1$**	**Hop$\mathcal{E}_0$**	**Gambit**	**Guile**	**Stklos**	**Bigloo**	**Racket**
almabench	10.42 (1.0 δ)	28.96 (2.78 δ)	***32.07*** (3.08 δ)	14.84 (1.42 δ)	17.36 (1.67 δ)	**0.77** (0.07 δ)	2.79 (0.27 δ)
bague	12.43 (1.0 δ)	27.16 (2.19 δ)	***43.24*** (3.48 δ)	9.76 (0.79 δ)	13.93 (1.12 δ)	**0.09** (0.01 δ)	0.82 (0.07 δ)
beval	10.22 (1.0 δ)	***20.39*** (2.0 δ)	15.6 (1.53 δ)	5.74 (0.56 δ)	8.13 (0.8 δ)	**0.14** (0.01 δ)	0.77 (0.08 δ)
boyer	9.95 (1.0 δ)	35.07 (3.52 δ)	***70.63*** (7.1 δ)	8.11 (0.82 δ)	19.88 (2.0 δ)	**0.41** (0.04 δ)	1.04 (0.1 δ)
conform	10.21 (1.0 δ)	18.95 (1.86 δ)	***23.28*** (2.28 δ)	7.84 (0.77 δ)	11.5 (1.13 δ)	**0.25** (0.02 δ)	1.28 (0.13 δ)
earley	10.21 (1.0 δ)	44.74 (4.38 δ)	***46.97*** (4.6 δ)	8.07 (0.79 δ)	21.53 (2.11 δ)	**1.22** (0.12 δ)	1.35 (0.13 δ)
fft	10.42 (1.0 δ)	19.41 (1.86 δ)	***27.1*** (2.6 δ)	7.46 (0.72 δ)	10.79 (1.04 δ)	**0.21** (0.02 δ)	1.01 (0.1 δ)
fib	11.09 (1.0 δ)	18.0 (1.62 δ)	***32.95*** (2.97 δ)	7.0 (0.63 δ)	8.02 (0.72 δ)	**0** (0.0 δ)	0.82 (0.07 δ)
leval	9.44 (1.0 δ)	24.89 (2.64 δ)	***35.4*** (3.75 δ)	8.34 (0.88 δ)	20.12 (2.13 δ)	**0.77** (0.08 δ)	2.36 (0.25 δ)
maze	11.01 (1.0 δ)	31.96 (2.9 δ)	56.32 (5.12 δ)	11.25 (1.02 δ)	***126.2*** (11.46 δ)	**0.51** (0.05 δ)	1.81 (0.16 δ)
mbrot	9.96 (1.0 δ)	20.37 (2.05 δ)	***23.58*** (2.37 δ)	10.65 (1.07 δ)	15.25 (1.53 δ)	**0.09** (0.01 δ)	0.41 (0.04 δ)
nucleic	10.01 (1.0 δ)	28.87 (2.88 δ)	***36.26*** (3.62 δ)	12.81 (1.28 δ)	17.46 (1.74 δ)	**0.58** (0.06 δ)	1.17 (0.12 δ)
peval	9.78 (1.0 δ)	15.18 (1.55 δ)	***23.63*** (2.42 δ)	7.5 (0.77 δ)	10.47 (1.07 δ)	**0.33** (0.03 δ)	1.1 (0.11 δ)
puzzle	9.73 (1.0 δ)	34.79 (3.58 δ)	***63.85*** (6.56 δ)	8.63 (0.89 δ)	23.31 (2.4 δ)	**0.21** (0.02 δ)	1.46 (0.15 δ)
qsort	10.38 (1.0 δ)	31.76 (3.06 δ)	***65.51*** (6.31 δ)	9.33 (0.9 δ)	26.75 (2.58 δ)	**0.2** (0.02 δ)	1.09 (0.11 δ)
queens	10.28 (1.0 δ)	22.32 (2.17 δ)	***42.6*** (4.14 δ)	-	12.9 (1.25 δ)	**0.42** (0.04 δ)	0.81 (0.08 δ)
sieve	10.14 (1.0 δ)	28.18 (2.78 δ)	***42.11*** (4.15 δ)	7.56 (0.75 δ)	12.28 (1.21 δ)	**1.09** (0.11 δ)	1.48 (0.15 δ)
traverse	10.31 (1.0 δ)	34.45 (3.34 δ)	***51.96*** (5.04 δ)	7.29 (0.71 δ)	16.06 (1.56 δ)	**0.74** (0.07 δ)	1.05 (0.1 δ)

Figure 29. Benchmarks timing of various Scheme implementations on Intel x86. Times (user + system) are expressed in seconds.

	Interpreters					*Compilers*
Bench	**Hop$\mathcal{E}_1$**	**Hop$\mathcal{E}_0$**	**Gambit**	**Guile**	**Stklos**	**Bigloo**
almabench	856.96 (1.0 δ)	1416.19 (1.65 δ)	***1477.31*** (1.72 δ)	991.47 (1.16 δ)	1212.87 (1.42 δ)	**418.36** (0.49 δ)
bague	194.85 (1.0 δ)	644.95 (3.31 δ)	***6850.4*** (35.16 δ)	180.56 (0.93 δ)	232.94 (1.2 δ)	**3.95** (0.02 δ)
beval	227.33 (1.0 δ)	***538.15*** (2.37 δ)	359.17 (1.58 δ)	128.39 (0.56 δ)	131.36 (0.58 δ)	**2.92** (0.01 δ)
boyer	216.07 (1.0 δ)	811.51 (3.76 δ)	***1425.14*** (6.6 δ)	179.95 (0.83 δ)	331.45 (1.53 δ)	**9.95** (0.05 δ)
conform	310.59 (1.0 δ)	568.61 (1.83 δ)	***587.37*** (1.89 δ)	161.14 (0.52 δ)	243.58 (0.78 δ)	**6.67** (0.02 δ)
earley	264.07 (1.0 δ)	***1108.18*** (4.2 δ)	1075.99 (4.07 δ)	164.92 (0.62 δ)	504.38 (1.91 δ)	**19.17** (0.07 δ)
fft	295.25 (1.0 δ)	***515.72*** (1.75 δ)	485.53 (1.64 δ)	196.06 (0.66 δ)	338.33 (1.15 δ)	**15.82** (0.05 δ)
fib	215.59 (1.0 δ)	450.61 (2.09 δ)	***631.94*** (2.93 δ)	153.38 (0.71 δ)	189.22 (0.88 δ)	**10.05** (0.05 δ)
leval	177.86 (1.0 δ)	575.73 (3.24 δ)	***616.69*** (3.47 δ)	167.96 (0.94 δ)	367.09 (2.06 δ)	**17.9** (0.1 δ)
maze	275.97 (1.0 δ)	827.46 (3.0 δ)	1220.07 (4.42 δ)	289.0 (1.05 δ)	***2580.67*** (9.35 δ)	**30.65** (0.11 δ)
mbrot	293.51 (1.0 δ)	***563.83*** (1.92 δ)	440.74 (1.5 δ)	286.17 (0.97 δ)	437.59 (1.49 δ)	**35.43** (0.12 δ)
nucleic	539.75 (1.0 δ)	1030.2 (1.91 δ)	***1180.33*** (2.19 δ)	536.86 (0.99 δ)	692.84 (1.28 δ)	**118.45** (0.22 δ)
peval	309.69 (1.0 δ)	389.27 (1.26 δ)	***481.86*** (1.56 δ)	163.95 (0.53 δ)	196.98 (0.64 δ)	**10.7** (0.03 δ)
puzzle	230.99 (1.0 δ)	823.45 (3.56 δ)	***1089.81*** (4.72 δ)	203.03 (0.88 δ)	395.71 (1.71 δ)	**4.78** (0.02 δ)
qsort	235.21 (1.0 δ)	771.1 (3.28 δ)	***1199.74*** (5.1 δ)	218.98 (0.93 δ)	601.8 (2.56 δ)	**3.98** (0.02 δ)
queens	203.65 (1.0 δ)	***518.23*** (2.54 δ)	-	-	232.02 (1.14 δ)	**10.95** (0.05 δ)
sieve	328.73 (1.0 δ)	746.74 (2.27 δ)	***908.24*** (2.76 δ)	232.44 (0.71 δ)	341.05 (1.04 δ)	**39.64** (0.12 δ)
traverse	250.07 (1.0 δ)	938.21 (3.75 δ)	***1245.06*** (4.98 δ)	190.03 (0.76 δ)	320.55 (1.28 δ)	**23.24** (0.09 δ)

Figure 30. Benchmarks timing of various Scheme implementations on Arm. Times (user + system) are expressed in seconds.

tion language of the interpreter might not be crucial for the performance, since Guile is implemented in C and HOP$\mathcal{E}_1$ in HOP.

We have compared the memory allocations of HOP$\mathcal{E}_1$ and HOP$\mathcal{E}_0$. The result is presented Figure 31. Note that for this experiment we have executed the tests with smaller input values for the tests. HOP$\mathcal{E}_1$ reduces significantly the allocated memory, benefiting from its stack structure (see Section 3.5). Greater amounts of allocated memory for 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 HOP$\mathcal{E}_0$ allocates 350 times more memory than HOP$\mathcal{E}_1$ for `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 `fft` but HOP$\mathcal{E}_1$ executes it twice faster. Other tests such as `puzzle` or `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 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 difference 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 HOP. We have used 5 test programs. First, `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 implemen-

	Allocated memory	
Bench	**Hop$\mathcal{E}_1$**	**Hop$\mathcal{E}_0$**
almabench	**13MB - 7**	***28MB - 28***
bague	**3MB - 1**	***71MB - 3***
beval	**7MB - 3**	***144MB - 69***
boyer	**5MB - 2**	***57MB - 30***
conform	**12MB - 9**	***46MB - 22***
earley	**14MB - 7**	***135MB - 87***
fft	**62MB - 76**	***84MB - 80***
fib	**3MB - 1**	***1050MB - 478***
leval	**68MB - 34**	***756MB - 347***
maze	**10MB - 3**	***92MB - 19***
mbrot	**51MB - 59**	***116MB - 90***
nucleic	**46MB - 23**	***130MB - 70***
peval	**32MB - 27**	***48MB - 23***
puzzle	**3MB - 1**	***263MB - 124***
qsort	**5MB - 2**	***128MB - 77***
queens	**114MB - 55**	***352MB - 97***
seive	**7MB - 3**	***27MB - 13***
traverse	**8MB - 4**	***316MB - 131***

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.

tations of all sort of programming languages. All implementations are free to use any technique or trick as long as they implement the same algorithm. A permissive understanding of this sentence gives a lot of freedom to the implementors. For instance, the PHP implementation of the `spectral-norm` uses two threads while all the other languages only use one. The idea of the Shootout game is to answer the following question: *for each considered problem, what is the fastest implementation possible for each tested system?* The Figure 32 presents the results on x86. Figure 33 presents them on Arm.

As a confirmation 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, HOP$\mathcal{E}_1$ delivers good performances for `fib`, `btrees`, and `snorm`, 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:

```
(let loop ((i 0))
  (set! Zi (+fl (*fl 2.0 (*fl Zr Zi)) Ci))
  (set! Zr (+fl (-fl Tr Ti) Cr))
  (set! Tr (*fl Zr Zr))
  (set! Ti (*fl Zi Zi))
  (set! k (+fx k 5))
  (when (<=fl (+fl 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 HOP$\mathcal{E}_1$ 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 HOP$\mathcal{E}_1$ 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 HOP$\mathcal{E}_1$.

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 HOP$\mathcal{E}_1$ 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.

HOP$\mathcal{E}_1$ has fulfilled its primary goal, which was to eliminate the major weaknesses of the former HOP$\mathcal{E}_0$ interpreter. As demonstrated in Section 5, HOP$\mathcal{E}_1$ has significantly exceeded that goal. It is one of the fastest Scheme interpreters we have tested. HOP$\mathcal{E}_1$ is between two and three times faster than HOP$\mathcal{E}_0$. 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 Courtès, Eric Gallesio, Marc Feeley, and Gérard Berry for their helpful feedback on this work.

	Interpreters							*Compilers*	
Bench	**Hop$\mathcal{E}_1$**	**Hop$\mathcal{E}_0$**	**Lua**	**Python**	**Ruby**	**Php**	**Perl**	**Bigloo**	**JS V8**
fib	7.38 (1.0 δ)	12.6 (1.71 δ)	8.85 (1.2 δ)	33.43 (4.53 δ)	9.6 (1.3 δ)	24.47 (3.32 δ)	***56.75*** (7.69 δ)	**0.0** (0.0 δ)	1.02 (0.14 δ)
btrees	10.81 (1.0 δ)	22.85 (2.11 δ)	16.64 (1.54 δ)	19.34 (1.79 δ)	9.92 (0.92 δ)	***34.45*** (3.19 δ)	31.73 (2.94 δ)	1.39 (0.13 δ)	**0.68** (0.06 δ)
fasta	10.08 (1.0 δ)	***32.88*** (3.26 δ)	1.98 (0.2 δ)	0.6 (0.06 δ)	5.56 (0.55 δ)	5.73 (0.57 δ)	14.25 (1.41 δ)	**0.07** (0.01 δ)	1.83 (0.18 δ)
mbrot	10.88 (1.0 δ)	***31.44*** (2.89 δ)	2.47 (0.23 δ)	8.12 (0.75 δ)	14.79 (1.36 δ)	4.72 (0.43 δ)	17.99 (1.65 δ)	**0.12** (0.01 δ)	1.91 (0.18 δ)
snorm	6.1 (1.0 δ)	25.31 (4.15 δ)	5.49 (0.9 δ)	25.24 (4.14 δ)	11.71 (1.92 δ)	11.11 (1.82 δ)	***25.39*** (4.16 δ)	**0.2** (0.03 δ)	0.35 (0.06 δ)

Figure 32. Benchmarks timing of various languages on Intel x86. Times (user + system) are expressed in seconds.

	Interpreters							*Compilers*
Bench	**Hop$\mathcal{E}_1$**	**Hop$\mathcal{E}_0$**	**Lua**	**Python**	**Ruby**	**Php**	**Perl**	**Bigloo**
fib	134.2 (1.0 δ)	346.54 (2.58 δ)	232.45 (1.73 δ)	-	215.73 (1.61 δ)	748.68 (5.58 δ)	***1114.83*** (8.31 δ)	**7.58** (0.06 δ)
btrees	226.17 (1.0 δ)	524.3 (2.32 δ)	-	-	231.92 (1.03 δ)	-	***721.41*** (3.19 δ)	**44.46** (0.2 δ)
fasta	413.18 (1.0 δ)	***946.75*** (2.29 δ)	95.8 (0.23 δ)	-	253.09 (0.61 δ)	244.09 (0.59 δ)	358.7 (0.87 δ)	**21.93** (0.05 δ)
mbrot	310.17 (1.0 δ)	***855.52*** (2.76 δ)	94.28 (0.3 δ)	-	549.04 (1.77 δ)	184.43 (0.59 δ)	510.2 (1.64 δ)	**46.4** (0.15 δ)
snorm	198.62 (1.0 δ)	596.01 (3.0 δ)	198.69 (1.0 δ)	444.52 (2.24 δ)	488.88 (2.46 δ)	-	***596.98*** (3.01 δ)	**52.44** (0.26 δ)

Figure 33. Benchmarks timing of various languages on Arm. Times (user + system) are expressed in seconds.

References

[1] J. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[2] C.-F. Bolz and A. Rigo. How to *not* write Virtual Machines for Dynamic Languages. July 2007.

[3] G. Boudol, Z. Luo, T. Rezk, and M. Serrano. Towards Reasoning for Web Applications: an Operational Semantics for Hop. In *Proceedings of the first Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications (APLWACA'10)*, Toronto, Canada, June 2010.

[4] S. Brunthaler. Efficient Interpretation using Quickening. In *Proceedings of the Second Dynamic Languages Symposium*, pages 1–14, 2010.

[5] S. Brunthaler. Interpreter instruction scheduling. In *Proceedings of the 14th International Conference on Compiler Construction (CC)*, pages 164–178, Mar. 2011.

[6] CPython development team. Python/ceval.c, Feb. 2011.

[7] N. De Bruijn. Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[8] A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[9] M. Feeley and G. Lapalme. Using closures for code generation. *Computer Languages*, 12:47–66, 1987.

[10] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M.-R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[11] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 7(11):1159–1176, 2005.

[12] R. Kelsey, W. Clinger, and J. Rees. The Revised(5) Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), Sept. 1998.

[13] P. Klint. Interpretation techniques. *Software — Practice & Experience*, 11(9):963–973, September 1981.

[14] F. Loitsch and M. Serrano. *Trends in Functional Programming*, volume 8, chapter Hop Client-Side Compilation, pages 141–158. Seton Hall University, Intellect Bristol (ed. Morazán, M. T.), UK/Chicago, USA, 2008.

[15] M. Mihocka. No execute! – The Common CPU Interpreter Loop Revisited, Sept. 2008.

[16] S. Pemberton and M. Daniels. *Pascal Implementation: the P4 Compiler and Interpreter*. Ellis Horwood, 1983.

[17] M. Serrano. HOP, a Fast Server for the Diffuse Web. In *Invited paper of the 11th international conference on Coordination Models and Languages (COORDINATION'09)*, Lisbon, Portugal, June 2009.

[18] M. Serrano and M. Feeley. Storage Use Analysis and its applications. In *1fst* ACM SIGPLAN *Int'l Conference on Functional Programming (ICFP)*, pages 50–61, Philadelphia, Penn, USA, May 1996.

[19] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon, USA, Oct. 2006.

[20] M. Serrano and C. Queinnec. A multi-tier semantics for Hop. *Higher Order and Symbolic Computation (HOSC)*, 2010.

[21] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 2(1):161–177, 1992.

[22] O. Waddell, D. Sarkar, and K. Dybvig. Fixing letrec: A faithful yet efficient implementation of scheme's recursive binding construct. *Higher-Order and Symbolic Computation*, 18(3-4):299–326, 2005.