Engineering Definitional Interpreters

Jan Midtgaard
Department of Computer Science
Aarhus University
jmi@cs.au.dk

Norman Ramsey
Department of Computer Science
Tufts University
nr@cs.tufts.edu

Bradford Larsen
Veracode
blarsen@veracode.com

Abstract

A definitional interpreter should be clear and easy to write, but it may run 4–10 times slower than a well-crafted bytecode interpreter. In a case study focused on implementation choices, we explore ways of making definitional interpreters faster without expending much programming effort. We implement, in OCaml, interpreters based on three semantics for a simple subset of Lua. We compile the OCaml to $x86$ native code, and we systematically investigate hundreds of combinations of algorithms and data structures. In this experimental context, our fastest interpreters are based on natural semantics; good algorithms and data structures make them 2–3 times faster than naïve interpreters. Our best interpreter, created using only modest effort, runs only 1.5 times slower than a mature bytecode interpreter implemented in C.

Categories and Subject Descriptors D.3.4 [*Processors*]: Interpreters

General Terms Algorithms, Languages, Performance, Theory

Keywords Interpreters, semantics, language implementation

1. Introduction

In early days, McCarthy (1960) defined LISP by writing an interpreter. Then Reynolds (1972) showed us that an interpreter is a leaky abstraction: the metalanguage can influence the semantics of the language we are trying to define. Eventually, we found better ways of defining languages, using formalisms from Strachey and Scott (Stoy 1977), Plotkin (1981), Kahn (1987), and Felleisen (1987). If you like declarative languages, you understand that in any of these styles, a definition can be translated pretty directly into an interpreter, which we might still call "definitional." Danvy (2006) even showed how relationships between semantics and interpreters can be derived through transformation.

But did you ever wonder if this kind of definitional interpreter could be more than just a toy? More than a teaching tool? If you could do more with it than just get insight? If you could write interesting programs, then use a definitional interpreter to run them? We did.

To address these questions, we created and studied 644 interpreters, which use a couple of dozen variations on each of 23 basic ideas, each of which is based on one of three language definitions. We estimated both performance and programming effort.

PPDP '13, September 16–18, 2013, Madrid, Spain.

http://dx.doi.org/10.1145/2505879.2505894

In detail, we make the following contributions:

- We investigate three styles of semantics, each of which leads to a family of definitional interpreters. A "family" is characterized by its *representation of control contexts*. Our best-performing interpreter, which arises from a natural semantics, represents control contexts using control contexts of the metalanguage.
- We evaluate other implementation choices: how names are represented, how environments are represented, whether the interpreter has a separate "compilation" step, where intermediate results are stored, and how loops are implemented.
- We identify combinations of implementation choices that work well together. For example, if you do not want to bother writing a distinct compilation step, you should intern each name in the abstract-syntax tree so that it has a constant-time equality test. This choice, together with a simple association list for local variables, performs as well as some interpreters that *do* have a distinct compilation step (Section 10).

Because our performance measurements use only OCaml 3.10.2 compiling to $x86$ native code, conclusions about performance are necessarily preliminary. But it might not be too hard to engineer a definitional interpreter that you would be willing to use in practice: our best-performing implementation is nearly as easy to create and change as a naïve implementation, yet on this one platform, it performs about 3 times better than a naïve definitional interpreter and only 1.5 times worse than a mature bytecode interpreter.

The value of this work lies in a systematic method of engineering interpreters. We focus on the path from semantics to implementation, on the most easily implemented ideas for reducing interpretive overhead, and on the most salient problems that arise.

- We begin with semantic starting points. A semantics typically describes a core calculus, but even microbenchmarks require a bigger programming language. Because we want to compare our implementations with a mature, production bytecode interpreter, we define μLua, a subset of Lua 2.5 (Section 2).
- We describe a common infrastructure shared by all our interpreters, including some fundamental representation choices which we arrange in a "string cube" (Section 3). Using this infrastructure, we present direct implementations of natural semantics, denotational semantics, and abstract machines, using the functional language OCaml (Section 4).
- We improve performance by adding a "compilation step" to two of our three definitional interpreters. A compilation step analyzes abstract syntax and looks up local variables just once, then returns a first-class function that can be applied (Section 5).

We quantify performance as we go along, and from the examples we present, you should be able to estimate programming effort. In Section 10, as another measure of programming effort, we show the sizes of the most interesting interpreters.

2. Experimental framework

Reynolds (1998) observes that a definitional interpreter is "stylistically similar" to a semantic definition. Our data show that simple changes to a definitional interpreter can improve average run-time performance by a factor of 4 or more. The improvement depends on what kind of semantic definition you start with. We explore natural semantics (Kahn 1987); abstract machines that use explicit contexts (Landin 1964; Felleisen, Findler, and Flatt 2009); and a denotational semantics that uses continuations (Stoy 1977). We considered a small-step structural operational semantics without explicit contexts (Plotkin 1981), but the time and effort required to find a redex deep in an abstract-syntax tree, then splice in a new subterm, seemed likely to be prohibitive.

We compare simple definitional interpreters with a mature bytecode interpreter. As the bytecode interpreter, we have chosen version 2.5 of the Lua programming language (Ierusalimschy, de Figueiredo, and Celes 2007). Among popular interpreted language implementations, including Perl, Python, and Ruby, Lua not only performs best (Bagley 2002) but also has the smallest implementation.

Lua 2.5 is mature enough to be taken seriously but small enough that writing multiple implementations (of a subset) is not an overwhelming chore. Lua 2.5 required three years to develop, was noticed in the academic literature, and was used for applications in the petrochemical industry (Ierusalimschy, de Figueiredo, and Celes 1996). Also, unlike Lua versions 3 and higher, Lua 2.5 does not have nested functions. We have avoided first-class, nested functions because all by themselves they would require a large study.

2.1 Summary of Lua 2.5

Lua 2.5 (hereafter just "Lua") is a dynamically typed language with six types: nil, string, number, function, table, and userdata. Nil is a singleton type containing only the value nil. A table is a mutable hash table in which any value except nil may be used as a key. Userdata is an opaque type which enables a client program to add new data abstractions to the interpreter. Except for table, all the built-in types are immutable; userdata may be mutable at the client's discretion (Ierusalimschy, de Figueiredo, and Celes 1996).

Lua's abstract syntax has three significant syntactic categories: top-level declaration, statement, and expression. Functions are declared only at top level; Lua 2.5 has first-class, non-nested functions.

A name stands for a mutable location containing a value. Lua is statically scoped; each name is bound either to a global variable or to a local variable of the statement sequence in which it appears. (A formal parameter of a function has the same status as a local variable of the function's body.) Each local variable must be explicitly declared and can be mapped to a location at compile time. Because the name space of global variables can be extended dynamically, referring to a global variable involves a run-time lookup.

Even Lua 2.5 is a relatively large language, and as part of our study we have implemented 23 core evaluators. To keep this task manageable, we have devised an even smaller language we call μLua (pronounced "micro-Lua"). Throughout this paper, we refer to μLua as our *object language*.

2.2 Syntax and semantics of μLua

μLua exists only to be the subject of experiments. Its role in our experimental study is analogous to the role of a core calculus in a theoretical study. Compared with Lua 2.5, μLua omits a little syntax (and/or/repeat), multiple return values, about 40 predefined functions, and Lua 2.5's *fallback* mechanism for capturing erroneous run-time events. The former two are expressible by desugaring and the latter two by modest extensions to the interpreter infrastructure. In addition, μLua evaluates arguments from left to right. (Lua does not specify an order.) μLua is expressive enough that we have been able to port most of Bagley's (2002) benchmarks. μLua's syntax is shown in Figure 1; its semantics are sketched below. (Our goal is not to give a complete semantics to μLua but to illustrate connections between semantics and implementations.)

program ⇒ {*declaration*}
declaration ⇒ *statement*
| `function` *lvalue* `(` [*name* {`,` *name*}] `)` {*statement*} `end`
statement ⇒ *lvalue* `=` *exp*
| `local` *name* `=` *exp*
| `while` *exp* `do` {*statement*} `end`
| `if` *exp* `then` {*statement*} `else` {*statement*} `end`
| `return` *exp*
| *exp* `(` [*exp* {`,` *exp*}] `)` -- Call
exp ⇒ *name*
| *value*
| *exp* `[` *exp* `]` -- Table access
| *exp* `(` [*exp* {`,` *exp*}] `)` -- Call
lvalue ⇒ *name* | *exp* `[` *exp* `]`

Figure 1. Concrete syntax of μLua

value ⇒ *number* | *nil* | *string* | *table* | *function*

Figure 2. Values of μLua

Our semantic techniques are standard. We model the state of a Lua interpreter as a global environment ξ, which maps a name to a mutable cell containing a value, plus a store σ, which gives the contents of each mutable cell. In addition, each activation of a Lua function has a local environment ρ, which maps names to mutable cells. There is also a "local" environment ρ_t associated with the top-level sequence of *declaration*s. Variables bound in ρ_t are visible only in declarations, not inside functions.

For simplicity, we model ξ as an infinite environment that maps each name to a distinct mutable cell, which in the initial σ contains nil. The semantics may then treat ξ as immutable. In an implementation, mutable cells are allocated to ξ on demand.

Figure 3 shows ρ and ξ and all the other metavariables we use in our semantics, as well as our list notation, which is ML notation. Figure 4 shows semantic functions, and Figures 5 and 6 show our first semantics, which is a natural semantics (Kahn 1987). Figure 1 is organized top-down, with statements before expressions, but both the semantics and later the code are easier to understand if we present expressions before statements.

The figures use these formal judgments:

- The judgment $\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$ means that if we evaluate expression e with local environment ρ and store σ, evaluation terminates, producing a value v and a new store σ'.
- The judgment $\langle ss, \rho, \sigma\rangle \Downarrow \sigma'$ means that if we evaluate a sequence of statements ss in local environment ρ and with store σ, evaluation terminates, producing a new store σ'.
- The judgment $\langle ss, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$ means that if we evaluate a sequence of statements ss in local environment ρ and with store σ, evaluation ends in a `return`, producing a value v and a new store σ'.

When possible, Figure 6 combines the statement forms: to avoid near-duplicate rules, we use the metavariable o to stand either for a termination outcome σ' or a `return` outcome $\langle v, \sigma'\rangle$.

x	A name
l	A location
e	An expression
$\bullet$	A hole (in an evaluation context)
v	A value
ss	A sequence of statements
ρ	An environment mapping local variables to locations
ξ	An environment mapping global variables to locations
σ	A machine state mapping locations to values
S	A stack of evaluation contexts (also $e :: S$)
o	An outcome of evaluating a statement: either σ or $\langle v, \sigma \rangle$
θ	A continuation mapping states to answers
κ	An expression continuation mapping values to continuations
κ_r	The return continuation
$s :: ss$	A cons cell: statement followed by statements
$ss \mathbin{@} ss'$	Appended lists
$[s_1, \ldots, s_n]$	A literal list

Figure 3. Metavariables and list notation

location	This function implements Lua's naming rule: $$location(x, \rho, \xi) = \begin{cases} \rho(x), & \text{if } x \in \operatorname{dom} \rho \\ \xi(x), & \text{if } x \notin \operatorname{dom} \rho \end{cases}$$
gettable	Function $gettable(v_t, v_k)$ looks up key v_k in table v_t. It is defined iff v_t is a table and $v_k \neq nil$.
fresh	In a denotational definition, $fresh(\sigma)$ deterministically identifies a location that is not in the domain of σ.

Figure 4. Semantic functions

$\boxed{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$

$$\frac{}{\langle x, \rho, \sigma \rangle \Downarrow \langle \sigma(location(x, \rho, \xi)), \sigma \rangle} \quad \text{(VAR)}$$

$$\frac{}{\langle v, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle} \quad \text{(LITERAL)}$$

$$\frac{\langle e_1, \rho, \sigma \rangle \Downarrow \langle v_t, \sigma'' \rangle \qquad v_t \text{ is a table} \qquad \langle e_2, \rho, \sigma'' \rangle \Downarrow \langle v_k, \sigma' \rangle \qquad v_k \neq nil}{\langle e_1\,\texttt{[}e_2\texttt{]}, \rho, \sigma \rangle \Downarrow \langle gettable(v_t, v_k), \sigma' \rangle} \quad \text{(TABLEELEMENT)}$$

$$\frac{\begin{array}{c}\langle e_f, \rho, \sigma \rangle \Downarrow \langle \texttt{function}(x_1, \ldots, x_n)\ ss_f\ \texttt{end}, \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \ \cdots \ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ l_i \notin \operatorname{dom} \sigma_n, \quad 1 \leq i \leq n \\ l_i \neq l_j, \quad 1 \leq i < j \leq n \\ \rho_f = \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\} \\ \sigma_f = \sigma_n\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\} \\ \langle ss_f \mathbin{@} [\texttt{return}\ nil], \rho_f, \sigma_f \rangle \Downarrow \langle v, \sigma' \rangle\end{array}}{\langle e_f\,\texttt{(}e_1\texttt{, ... ,}e_n\texttt{)}, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad \text{(CALLEXP)}$$

Figure 5. Natural semantics of some μLua expressions (Rules for infix binary operators are omitted.)

As an alternative to the natural semantics in Figures 5 and 6, we considered small-step structural operational semantics. A classic small-step semantics rewrites abstract-syntax trees in place. We believe that such a semantics is both tedious to implement and likely to perform badly, so we have not investigated this alternative. But a small-step semantics can also be written as an abstract machine, like the CESK machine, which holds the context of evaluation in a data structure (Felleisen, Findler, and Flatt 2009, Part I). The reduction rules in Figure 7 are inspired by this machine.

$\boxed{\langle ss, \rho, \sigma \rangle \Downarrow \sigma'}$ $\boxed{\langle ss, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$

(To stand for a store σ' or a pair $\langle v, \sigma' \rangle$, we use metavariable o.)

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \qquad \langle ss, \rho, \sigma'\{location(x, \rho, \xi) \mapsto v\} \rangle \Downarrow o}{\langle x\ \texttt{=}\ e :: ss, \rho, \sigma \rangle \Downarrow o} \quad \text{(ASSIGN)}$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \quad l \notin \operatorname{dom} \sigma' \quad \langle ss, \rho\{x \mapsto l\}, \sigma'\{l \mapsto v\} \rangle \Downarrow o}{\langle \texttt{local}\ x\ \texttt{=}\ e :: ss, \rho, \sigma \rangle \Downarrow o} \quad \text{(LOCAL)}$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \qquad v \neq nil \qquad \langle ss', \rho, \sigma' \rangle \Downarrow \langle v', \sigma'' \rangle}{\langle (\texttt{while}\ e\ \texttt{do}\ ss'\ \texttt{end}) :: ss, \rho, \sigma \rangle \Downarrow \langle v', \sigma'' \rangle} \quad \text{(WHILERETURN)}$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \qquad v \neq nil \qquad \langle ss', \rho, \sigma' \rangle \Downarrow \sigma'' \qquad \langle (\texttt{while}\ e\ \texttt{do}\ ss'\ \texttt{end}) :: ss, \rho, \sigma'' \rangle \Downarrow o}{\langle (\texttt{while}\ e\ \texttt{do}\ ss'\ \texttt{end}) :: ss, \rho, \sigma \rangle \Downarrow o} \quad \text{(WHILETRUE)}$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \qquad v = nil \qquad \langle ss, \rho, \sigma' \rangle \Downarrow o}{\langle (\texttt{while}\ e\ \texttt{do}\ ss'\ \texttt{end}) :: ss, \rho, \sigma \rangle \Downarrow o} \quad \text{(WHILEFALSE)}$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \texttt{return}\ e :: ss, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad \text{(RETURN)}$$

$$\frac{}{\langle [], \rho, \sigma \rangle \Downarrow \sigma} \quad \text{(BLOCKEND)}$$

Figure 6. Natural semantics of some μLua statements (Rules for `if`, calls, and assignments to tables are omitted.)

$\boxed{\langle e \text{ or } v, S, \rho, \sigma \rangle \to \langle e' \text{ or } v', S', \rho', \sigma' \rangle}$

$$\begin{array}{rl} & \langle x, S, \rho, \sigma \rangle \to \langle \sigma(location(x, \rho, \xi)), S, \rho, \sigma \rangle \\ \langle e_1\,\texttt{[}e_2\texttt{]}, S, \rho, \sigma \rangle \to & \langle e_1, \bullet\texttt{[}e_2\texttt{]} :: S, \rho, \sigma \rangle \\ & \langle v, [], \rho, \sigma \rangle \to v \\ \langle v, \bullet\texttt{[}e\texttt{]} :: S, \rho, \sigma \rangle \to & \langle e, v\texttt{[}\bullet\texttt{]} :: S, \rho, \sigma \rangle \\ \langle v_k, v_t\texttt{[}\bullet\texttt{]} :: S, \rho, \sigma \rangle \to & \langle v, S, \rho, \sigma \rangle \quad \text{where } v = gettable(v_t, v_k) \end{array}$$

Figure 7. Abstract machine transitions for some μLua expressions (Rules for infix binary operators and for calls are omitted.)

Finally, Figure 8 shows a denotational semantics that uses Scott and Strachey's approach (Stoy 1977). A sequence of statements, in context, denotes a function from continuations to continuations. (One might prefer to specify a single statement as the basic construct and to define the denotation of a sequence by composition, but the semantics of μLua statements is not compositional: a `local` statement extends the environment ρ and so affects the context in which the meanings of succeeding statements are determined.)

We have implemented all of the semantics above using the functional language OCaml, which is our *metalanguage*. The techniques we use should be easily portable to any metalanguage that has first-class, nested functions and call-by-value semantics. As noted by Reynolds (1972), using a call-by-name metalanguage to implement a call-by-value object language requires more care.

$$\vec{\mathcal{S}}[\![x \texttt{ = } e :: ss]\!]\rho\xi\kappa_r\theta = \mathcal{E}[\![e]\!]\rho\xi(\lambda v.\lambda\sigma.\vec{\mathcal{S}}[\![ss]\!]\rho\xi\kappa_r\theta(\sigma\{location(x,\rho,\xi) \mapsto v\}))$$

$$\vec{\mathcal{S}}[\![\texttt{local } x \texttt{ = } e :: ss]\!]\rho\xi\kappa_r\theta = \mathcal{E}[\![e]\!]\rho\xi(\lambda v.\lambda\sigma.\vec{\mathcal{S}}[\![ss]\!]\rho\{x \mapsto \ell\}\xi\kappa_r\theta(\sigma\{\ell \mapsto v\})), \quad \text{where } \ell = fresh(\sigma)$$

$$\vec{\mathcal{S}}[\![\texttt{while } e \texttt{ do } ss' \texttt{ end} :: ss]\!]\rho\xi\kappa_r\theta = \mathbf{fix}(\lambda\theta'.\mathcal{E}[\![e]\!]\rho\xi(\lambda v.\mathbf{if}\ v \neq nil\ \mathbf{then}\ \vec{\mathcal{S}}[\![ss']\!]\rho\xi\kappa_r\theta'\ \mathbf{else}\ \vec{\mathcal{S}}[\![ss]\!]\rho\xi\kappa_r\theta))$$

$$\vec{\mathcal{S}}[\![\texttt{return } e :: ss]\!]\rho\xi\kappa_r\theta = \mathcal{E}[\![e]\!]\rho\xi\kappa_r$$

$$\vec{\mathcal{S}}[\![[\,]]\!]\rho\xi\kappa_r\theta = \theta$$

$$\mathcal{E}[\![x]\!]\rho\xi\kappa = \lambda\sigma.\kappa(\sigma(location(x,\rho,\xi)))\sigma$$

$$\mathcal{E}[\![v]\!]\rho\xi\kappa = \kappa v$$

$$\mathcal{E}[\![e_1\,\texttt{[}e_2\texttt{]}]\!]\rho\xi\kappa = \mathcal{E}[\![e_1]\!]\rho\xi(\lambda v_t.\mathcal{E}[\![e_2]\!]\rho\xi(\lambda v_k.\kappa(gettable(v_t, v_k))))$$

Figure 8. Continuation semantics of some μLua statements and expressions

2.3 Experimental method: Choice guided by measurement

By "engineering" a definitional interpreter, we mean choosing algorithms and data structures in a way that is guided by measurements. Measurement requires both a benchmark and a working interpreter, which embody many choices. To quantify the effects of any one choice, we average over the others.

Adding running times of different benchmarks is not meaningful, so to average, we use the geometric mean of ratios of running times. The geometric mean of n ratios is the nth root of their product. It is also the exponent of the arithmetic mean of the logarithms of the ratios.

To quantify the variation introduced by averaging, we use *geometric standard deviation*. This number is the exponent of the ordinary standard deviation of the logarithms of a population. Using an ordinary standard deviation, two-thirds of normally distributed data fall within a *distance* of one standard deviation from the mean. Using a geometric standard deviation, two-thirds of normally distributed data fall within a *factor* of one standard deviation from the mean.

For example, the first choice we present below is what data structure to use to represent each name: an atom or a string? On average, atoms are 1.57 times faster, and the geometric standard deviation is 1.54. Therefore, if the data are normally distributed, then in two-thirds of the other choices, comparing atoms to strings yields a ratio of running times between 1.02 and 2.42. (In practice, the data are *not* normally distributed, but the geometric standard deviation is still an effective way to quantify variation.)

Ordinary standard deviations can be reported compactly using the $\pm$ operator. We have not found a standard, compact notation for reporting geometric standard deviations—but we need one. Instead of saying "atoms are 1.57 times faster than strings, with a geometric standard deviation of 1.54," we therefore report that atoms are $1.57 \stackrel{\times}{\div} 1.54$ times faster. (We also use $\stackrel{\times}{\div}$ with percentages.)

When are the results of a measurement statistically significant? To judge significance, we use a paired t-test on logarithms of ratios. The interpretation of t depends on the number of measurements, but even for our smallest experiments a t value of 2.18 or more indicates at least 95% confidence that the measured effect is not the result of noise or random error. Additional explanations and tables containing the results of many paired t-tests are available at `http://www.cs.tufts.edu/~nr/interps.html`.

3. Representing semantic concepts

To create an interpreter from a semantics, we must choose representations of semantic concepts. Some concepts have one clearly optimal representation; others have several seemingly plausible potential representations. Where we found plausible choices, we evaluated them experimentally. The most pervasive choices about representation have to do with strings and names; three independent choices form a *string cube*, which is shown in Figure 9.

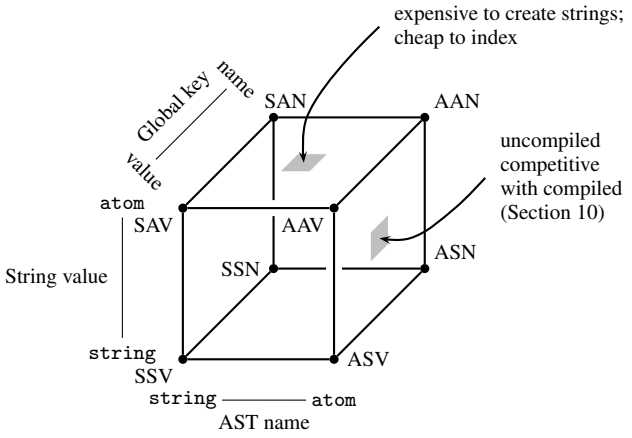

Figure 9. The string cube: representing names, strings, and ξ (As described in the text, a vertex's name encodes three choices.)

3.1 Names, strings, and the global environment

Each leaf of an abstract-syntax tree is either a name or a literal object-language value. The simplest representation of a name is a string. But an important alternative is an *atom*. An atom is a representation of strings that offers a constant-time equality test. Converting a string to an atom, called *interning*, requires a lookup in a data structure, typically a hash table. Representing names as atoms costs more at parse time, but speeds name lookup at interpretation time, especially if the environment ρ is represented by an association list. In Figure 9, the left face of the string cube shows names represented by strings, and the right face shows names represented by atoms. Even on the very worst benchmarks, atoms are never more than 12% slower than strings. On average, atoms are $1.57 \stackrel{\times}{\div} 1.54$ times faster ($t = 35.25$). The advantage conferred by atoms depends on the style in which the interpreter is written. Using atoms, the simple interpreters in Section 4 run $1.74 \stackrel{\times}{\div} 1.50$ times faster ($t = 22.18$); interpreters that have a "compilation step" (Section 5) run only $1.08 \stackrel{\times}{\div} 1.24$ times faster ($t = 6.09$).

Object-language string values may also be represented either by metalanguage strings or by atoms. In Figure 9, the bottom face of

the string cube shows μLua strings represented by OCaml strings, and the top face shows μLua strings represented by atoms. To three significant digits, the average performance of the two faces is identical; whatever difference exists is not statistically significant ($t = 0.15$). But depending on the choice of benchmark, there is a lot of variation ($\stackrel{\times}{\div} 1.5$). The top face is expensive for benchmarks that create many object-language strings but do not reuse them. The top face is cheap for benchmarks that use a few string values to index into many tables. These results are as we expect: If a μLua string is represented by an atom, it costs more to create, but its hash code is stored as part of the atom, so it is cheap to use as an index. If a μLua string is represented by an OCaml string, then every time the string is used as an index, its hash code is recomputed.

Given finite maps, implementing atoms requires little programming effort; our interface and implementation total 30 lines of OCaml.

The third choice on the string cube is more subtle: the representation of the keys that index the global environment ξ. Full Lua indexes ξ using general object-language values, as shown on the front face of the string cube. But μLua, like most other languages, uses only *names* to index into ξ. We can eliminate a level of indirection by using a representation of ξ which admits *only* of indexing by names and does not admit of indexing by more general values. This choice is shown on the back face of the string cube. The effect of this choice is small but statistically significant: on average, values are $1.02 \stackrel{\times}{\div} 1.28$ times faster than names ($t = 2.72$).

We refer to each vertex of the string cube using a three-character code: for example, the back lower-right vertex, which uses atoms to represent names, OCaml strings to represent μLua strings, and μLua names to index into ξ, is coded ASN. As another example, the choices made in Lua 2.5, which uses atoms for strings and names, and which uses general values to index into ξ, are coded AAV. Averaged over all interpreters and benchmarks, vertex AAV is $2.54 \stackrel{\times}{\div} 1.58$ times slower than Lua 2.5 ($t = 34.27$). The best vertex is ASN, which is $2.39 \stackrel{\times}{\div} 1.93$ times slower than Lua 2.5 ($t = 22.49$); the worst is SAN, which is $4.47 \stackrel{\times}{\div} 2.21$ times slower than Lua 2.5 ($t = 31.98$).

Figure 9 hides another implementation choice: a *strong* atom persists for the lifetime of the program, but a *weak* atom may be garbage collected if only the atom data structure points to it. When used to represent *names*, strong and weak atoms perform the same on average ($t = 1.89$, indicating no statistically significant difference). When used to represent *strings*, strong atoms run $1.06 \stackrel{\times}{\div} 1.13$ times faster than weak atoms ($t = 10.64$).

Having strong and weak atoms adds six more vertices to Figure 9, raising the number of representation choices to 14. (The two vertices on the lower left edge, which use no atoms, are unaffected.) For each choice, we replicated each interpreter described below.

3.2 Metalanguage and object-language functions

All interpreters share one external interface; each interpreter exports this function:

```
val func : name list  (* formal parameters *)
        -> stmt list  (* body *)
        -> globals    (* global names & state *)
        -> value list (* actual parameters *)
        -> value      (* result *)
```

Our infrastructure partially applies `func` to syntax and `globals`, then applies the metalanguage `Function` constructor (Figure 11) to the resulting metalanguage value of type `value list -> value`. The resulting `value` is stored in the global environment and reused every time the function is called.

To apply an object-language function, we call

```
val to_func : value -> (value list -> value)
```

which strips the `Function` constructor off its argument. Similarly, to apply a binary operator, we call

```
val binop : op -> (value -> value -> value)
```

3.3 State and environments

A machine state σ has only one efficient representation: we store the mutable parts of μLua's machine state in mutable reference cells and arrays provided by OCaml. Representing object-language state using an explicit functional data structure in the metalanguage is possible, but it would add a level of indirection and would require us to implement a garbage collector, decreasing performance and increasing programming effort.

Unlike the state, an environment ρ has more than one plausible representation. Our semantics use an immutable association list, which we implement as a value of type `ListEnv.env`. The `ListEnv` interface also exports operations that manipulate a mutable ξ of type `globals`, as well as the (implicit) object-language state σ:

```
module ListEnv : sig
  type env = (name * value ref) list
  val from_lists : name list -> value list -> env
  val lookup : name -> env -> globals -> value
  val extend : name -> value -> env -> env
  val rebind : name -> value -> env -> globals -> unit
end
```

Function `from_lists` creates a new environment which binds each name to a newly allocated cell that is initialized with the corresponding value. The other functions relate to the semantics as follows:

`lookup` $x\ \rho\ \xi = \sigma(location(x, \rho, \xi))$
`extend` $x\ v\ \rho = \rho\{x \mapsto \ell\}$, updating $\ell := v$, where $\ell = fresh(\sigma)$
`rebind` $x\ v\ \rho\ \xi$ mutates σ by $\rho(x) := v$ or $\xi(x) := v$

Environments can be represented using many other data structures. In Section 4.4, we experiment with mutable hash tables. We distinguish the representations by using the name IMMRHO for an interpreter that uses association lists and MUTRHO for one that uses hash tables.

3.4 Tables

Lua's central data structure is the mutable hash table, and the performance of hash tables determines the performance of many Lua benchmarks. For the fairest possible comparison, we have tried to replicate Lua's hash-table implementation in OCaml. The replica is not completely faithful; for example, the original uses an array of unboxed pairs, but because OCaml cannot represent an array of unboxed pairs, the replica uses instead a pair of arrays, each of which contains unboxed scalars.

As a sanity check, we compared our replica with OCaml's native implementation. Averaged over all interpreters and all vertices of the string cube, OCaml's native hash table is $1.02 \stackrel{\times}{\div} 1.22$ times slower ($t = 7.79$). We conclude that using the replica is reasonable.

4. Simple definitional interpreters

Unlike names, strings, tables, and environments, the abstract syntax and values of μLua have canonical representations: algebraic data types. Our representations are shown in Figures 10 and 11. The `Binop` constructor subsumes the table-indexing expression shown in Figure 1, as well as the other binary operators defined in type `op`.

```
type name (* representation varies *)
type stmt =
  | Assign   of lval * exp
  | WhileDo  of exp * stmt list
  | If       of exp * stmt list * stmt list
  | Return   of exp
  | Callstmt of exp * exp list
  | Local    of name * exp
and lval =
  | Lvar     of name
  | Lindex   of exp * exp    (* table element *)
and exp =
  | Var      of name
  | Lit      of value
  | Binop    of exp * op * exp
  | Call     of exp * exp list
and op =
  | Plus | Minus | Times | Div | Pow
  | Lt | Le | Gt | Ge | Eq | Ne
  | Concat | Not | Index
```

Figure 10. Representation of μLua's abstract syntax

```
type table      (* mutable hash table; rep. varies *)
type lua_string (* immutable string; rep. varies *)
type value =
  | Nil
  | Number   of float
  | String   of lua_string
  | Function of value list -> value
  | Table    of table
and globals = table
```

Figure 11. Representation of μLua's values

Both expressions and statements are evaluated in a context. We are particularly interested in the *control* context, which tells where an expression returns its value or what happens after a statement is evaluated. The choices available for representing control contexts depend on the semantics from which an interpreter is derived:

- An interpreter derived from a natural semantics works by structural recursion over abstract-syntax trees. The control context of an expression or statement is represented by a control context in the metalanguage—in other words, by the call stack in the implementation language.
- An interpreter derived from an abstract machine runs a loop, or more exactly, a pair of tail-recursive functions which keep the abstract-machine state in parameters. The control context of an expression or a statement is represented by a data structure which is allocated on the heap.
- An interpreter derived from a continuation semantics uses continuation-passing style (Reynolds 1972). Every control context is represented by a first-class function.

We can also "hybridize" an interpreter by using one representation for the control context of a statement and another for the control context of an expression. Because hybridization triples the number of potential interpreters, we hybridize only selected interpreters, based on performance (Section 7).

To build a simple interpreter, then, we choose a representation of control contexts, a representation of environments, and a vertex of the string cube (representation of names, representation of strings, representation of keys in ξ).

4.1 Using control contexts from the metalanguage

Our simplest interpreter uses structural recursion over abstract syntax; to represent each control context in the object language, the interpreter uses a control context of the metalanguage. The interpreter is named METACON-IMMRHO. Figures 12 to 14 show the types and implementations of the metalanguage functions that interpret μLua expressions, bodies, and functions.[1] In `exp`, the context of every recursive call is implicit in the metalanguage; the context is the right-hand side of a `let` binding. (One of those bindings is in the library function `List.map`.) Function `exp` should remind you of Figure 5: if `exp` $e\ \rho\ \xi$ is evaluated in metalanguage machine state σ, it returns value v if and only if $\langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$, where σ' represents the metalanguage machine state after evaluation. (In the semantics, we leave ξ implicit, but in the implementation, ξ must be explicit.)

```
  val exp  : exp       -> locals -> globals -> value
  val body : stmt list -> locals -> globals -> unit
```

Figure 12. METACON-IMMRHO type signatures

```
let rec exp e rho xi = match e with
  | Lit v -> v
  | Var x -> ListEnv.lookup x rho xi
  | Call (e, es) ->
      let v  = exp e rho xi in
      let vs = List.map (fun e -> exp e rho xi) es in
      to_func v vs
  | Binop (e1, op, e2) ->
      let v1 = exp e1 rho xi in
      let v2 = exp e2 rho xi in
      binop op v1 v2
```

Figure 13. METACON-IMMRHO expression interpreter

```
exception Returnx of value
let rec body ss rho xi =
  let rec stmts = function
  | [] -> ()
  | s :: ss -> match s with
    | Assign (Lvar x, e) ->
        let v = exp e rho xi in
        ListEnv.rebind x v rho xi;
        stmts ss
    | WhileDo (e, b) ->
        while not_nil (exp e rho xi) do stmts b done;
        stmts ss
    | If (e, t, f) ->
        if not_nil (exp e rho xi) then stmts t
        else stmts f;
        stmts ss
    | Return e -> raise (Returnx (exp e rho xi))
    | Local (x, e) ->
        let v    = exp e rho xi in
        let rho' = ListEnv.extend x v rho in
        body ss rho' xi
    (* other cases not shown *)
  in stmts ss

let func formals ss xi actuals =
  try
    body ss (ListEnv.from_lists formals actuals) xi; Nil
  with Returnx v -> v
```

Figure 14. METACON-IMMRHO statement & function interpreters

In `exp`, OCaml's `List.map` determines the order of evaluation of a μLua `Call`. Attending to this kind of detail requires programming effort that cannot be measured by counting lines of code.

In `body`, which in turn is implemented by `stmts`, we interpret statements (Figure 14). The code should remind you of Figure 6.

[1] Except for Figure 11, every code figure in this paper is automatically extracted from our source code. Qualifications of some names are removed.

```
type econtext
type scontext
val exp  : exp -> econtext -> locals -> globals -> value
val body :
  stmt list -> locals -> globals -> scontext -> value
```

Figure 15. DATACON-IMMRHO type signatures

```
type eframe =
  | BinLeft  of op * exp
  | BinRight of value * op

let rec exp e stk rho xi =
 let rec eval e stk = match e with
 | Var x -> produce (ListEnv.lookup x rho xi) stk
 | Binop (e1, op, e2) -> eval e1 (BinLeft (op, e2) :: stk)
 and produce v = function
 | [] -> v
 | BinLeft (op,e)  :: stk -> eval e (BinRight (v,op) :: stk)
 | BinRight(v1,op) :: stk -> produce (binop op v1 v) stk
 in eval e []
```

Figure 16. DATACON-IMMRHO expression interpreter

```
type sframe =
  | Block of stmt list
  | Env   of locals

let rec body ss rho xi =
 let exp = fun e -> exp e [] rho xi in
 let rec stmts ss stk = match ss with
 | [] -> (match stk with
          | [] -> Nil
          | Block ss   :: stk -> stmts ss stk
          | Env oldrho :: stk -> body [] oldrho xi stk)
 | s :: ss -> match s with
   | WhileDo (e, b) ->
       if not_nil (exp e) then
         stmts b (Block (s::ss) :: stk)
       else
         stmts ss stk
   | Return e -> exp e
   | Local (x, e) ->
       let v    = exp e in
       let rho' = ListEnv.extend x v rho in
       body ss rho' xi (Env rho :: stk)
   (* other cases not shown *)
 in stmts ss
```

Figure 17. DATACON-IMMRHO statement interpreter

When the judgment $\langle ss, \rho, \sigma\rangle \Downarrow \sigma'$ is derivable, `stmts` ss changes the metalanguage state from σ to σ' and executes a metalanguage return. When the judgment $\langle ss, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$ is derivable, `stmts` ss changes the metalanguage state from σ to σ' and raises the metalanguage exception `Returnx` v. Each recursive call to `stmts` provides an appropriate control context for the ordinary return, when evaluation should continue with the next statement. Function `func` sets up a metalanguage exception handler which provides the proper control context for `Return`.

When names and string values are represented as strings, which is the simplest implementation, this interpreter runs $4.44 \stackrel{\times}{\div} 2.12$ times slower than Lua 2.5 ($t = 7.17$). When names in the abstract syntax are represented as atoms, performance doubles: on average, the interpreter is only $2.19 \stackrel{\times}{\div} 1.84$ times slower than Lua 2.5 ($t = 4.64$).

4.2 Using control contexts represented by data structures

Figures 15 to 17 show fragments of an interpreter that represents object-language control contexts as metalanguage data structures. The interpreter is named DATACON-IMMRHO. Figure 16 shows an implementation of the abstract machine in Figure 7. Function `eval` deals with machine states $\langle e, S, \rho, \sigma\rangle$ in which the first element is abstract syntax; function `produce` deals with machine states $\langle v, S, \rho, \sigma\rangle$ in which the first element is a value.

On average, a DATACON interpreter runs $10\% \stackrel{\times}{\div} 1.07$ slower than its corresponding METACON interpreter ($t = 19.92$)—and it requires more than twice as much code. Figures 16 and 17 show only half of the contexts (constructors of type `eframe`) used to evaluate expressions; our implementation uses four contexts for expressions and two for statements. And our code does *not* use explicit representations of the six contexts in which an expression appears inside a statement; because the common infrastructure we set up in Section 3 represents object-language functions as metalanguage functions of type `value list -> value`, we cannot pass an explicit representation of a calling context as a parameter.

Eliminating the expression-in-statement contexts does not compromise the advantages of explicit contexts. These advantages accrue when implementing control operations like exceptions or continuations (Felleisen 1987) and when finding roots for garbage collection. Since μLua's only control operator is `return`, no μLua program can transfer control between calling contexts. And as noted above, our interpreters are garbage collected by the metalanguage. Making all the contexts available as metalanguage data structures would mean extra work and no implementation advantage.

4.3 Using control contexts represented by functions

Our third and final simple interpreter, in Figures 18 to 20, represents each control context as a metalanguage function; it follows the continuation semantics sketched in Figure 8. The interpreter is named FUNCON-IMMRHO.

Although the state σ is implemented using the mutable state of the metalanguage, we define the type of continuations `'a cont` using an abstract type `state`, and we pass values of that type. Without arguments of `state` type, it is too easy to get order of evaluation wrong. For example, `sigma` controls order of evaluation for the cases of `Var` in Figure 19 and `Assign` in Figure 20. Type `state` is defined to be `unit`, so `sigma` is passed without run-time overhead.

Because the FUNCON-IMMRHO interpreter represents control contexts as functions, a context can be introduced with a lambda and eliminated by function application. There is no need for datatype declarations, named constructors, or case analysis; a full and faithful implementation of the semantics is almost effortless.

The FUNCON-IMMRHO interpreter is the worst-performing of the naïve definitional interpreters, performing $16\% \stackrel{\times}{\div} 1.09$ slower than the fastest one, METACON-IMMRHO ($t = 23.38$). We believe the problem is with allocation—dynamically, almost every call to `exp` requires the allocation of a fresh continuation closure.

4.4 Changing the representation of local environments

We do not expect an immutable association list to be an efficient data structure for looking up names. In OCaml, the standard finite map provided by the OCaml library is a mutable hash table. Lua 2.5 also uses mutable hash tables. Accordingly, we modified each of the interpreters above to use mutable hash tables. The modified variants are called METACON-MUTRHO, DATACON-MUTRHO, and FUNCON-MUTRHO. In each interpreter, the MUTRHO variant differs from the IMMRHO variant at only three points: the implementations of `Var`, `Assign`, and `Local`.

The new implementations of `Var` and `Assign` do exactly the same thing as the old ones, only using a different abstraction of environments. The new implementations of `Local` have more work to do: unlike the immutable environments of the IMMRHO interpreters,

```
    type state
    type 'a cont = state -> 'a
    type 'a kont = value -> 'a cont
    val exp  : exp       -> locals -> globals
          -> 'a kont -> 'a cont
    val body : stmt list -> locals -> globals
          -> 'a kont -> 'a cont -> 'a cont
```

Figure 18. FUNCON-IMMRHO type signatures

```
let exp e rho xi k =
  let rec eval e k = match e with
  | Lit v -> k v
  | Var x ->
      (fun sigma -> k (ListEnv.lookup x rho xi) sigma)
  | Binop (e1, op, e2) -> eval e1 (fun v1 ->
                            eval e2 (fun v2 ->
                              k (binop op v1 v2)))
  (* case for Call not shown *)
in eval e k
```

Figure 19. FUNCON-IMMRHO expression interpreter

```
let rec body ss rho xi k_r =
  let eval e k = exp e rho xi k in
  let rec stmt ss theta = match ss with
  | [] -> theta
  | Assign (Lvar x, e) :: ss ->
      eval e (fun v sigma ->
        ListEnv.rebind x v rho xi;
        stmt ss theta sigma)
  | WhileDo (e, b) :: ss ->
      fix (fun theta' ->
            eval e (fun v ->
              if not_nil v
              then stmt b theta'
              else stmt ss theta))
  | Return e :: ss -> eval e k_r
  | Local (x, e) :: ss ->
      eval e (fun v ->
        body ss (ListEnv.extend x v rho) xi k_r theta)
  (* other cases not shown *)
in stmt ss
```

Figure 20. FUNCON-IMMRHO statement interpreter

the mutable hash tables are not copied. Instead, when a `Local` declaration introduces a new binding for x, the code mutates ρ in place. Additional code associated with a `Local` declaration ensures that when x goes out of scope, its previous value is restored.

Mutable environments make the interpreters run $1.84 \stackrel{\times}{\div} 1.33$ times slower than immutable environments ($t = 28.65$). Profiling does not reveal an obvious reason, but the MUTRHO versions make 30% more calls to primitive equality, and they make 25% more calls to `caml_page_table_lookup` in the memory-management subsystem. Because standard profiling tools do not cope well with mutual recursion (Spivey 2004), it is difficult to track the sources of the calls, but the hash table is the obvious suspect.

5. Interpretation after a compilation step

Each simple interpreter above does case analysis of abstract syntax and lookup of local variables every time a function is called or the body of a loop is executed. Our first performance improvement is to stage the computation so that when we apply `func` (page 1005) to syntax and ξ, all case analysis and local-variable lookup is done immediately, and the resulting anonymous function of type `value list -> value` does not repeat that overhead. We call the first stage of the computation the *compilation step*; we call the second stage the *run step*.

```
type locals        = value array
type compiled_exp  = locals -> value
type compiled_stmt = locals -> unit
val exp  : exp       -> env -> globals -> compiled_exp
val body : stmt list -> env -> globals -> compiled_stmt
```

Figure 21. METACON-LETTMP type signatures

```
let rec exp e rho xi = match e with
| Lit v -> (fun sigma -> v)
| Var x -> (match lookup x rho with
           | Global  -> let s = global_key_of_name x in
                        (fun sigma -> getenvtable xi s)
           | Local n -> (fun sigma -> getlocal sigma n))
| Binop (e1, op, e2) ->
    let eval1   = exp e1 rho xi in
    let eval2   = exp e2 rho xi in
    let meta_op = binop op in
    (fun sigma -> let v1 = eval1 sigma in
                  let v2 = eval2 sigma in
                  meta_op v1 v2)
(* case for Call not shown *)
```

Figure 22. METACON-LETTMP compile & run steps (expressions)

```
let body ss rho xi =
 let rec stmts ss rho = match ss with
 | [] -> (fun sigma -> ())
 | s :: ss ->
     let eval_s  = stmt s rho in
     let eval_ss = stmts ss (newrho rho s) in
     (fun sigma -> eval_s sigma; eval_ss sigma)
 and stmt s rho =
   let eval e = exp e rho xi in
   match s with
   | WhileDo (e, b)  ->
       let evale = eval e in
       let evalb = stmts b rho in
       (fun sigma ->
        while not_nil (evale sigma) do evalb sigma done)
   | Local (x, e)    ->
       let eval_e = eval e in
       let n      = List.length rho in
       (fun sigma -> setlocal sigma n (eval_e sigma))
   (* other cases not shown *)
 in stmts ss rho
```

Figure 23. METACON-LETTMP compile & run steps (statements)

In every interpreter, we use the same compilation strategy:

- In the run step, every formal parameter and local variable is stored in an array (type `locals` in Figure 21).
- In the compilation step, every formal parameter and local variable is mapped, by type `env`, to an integer position in the local array.
- Function `func` allocates the local array for an activation, and it initializes the array with the values of the actual parameters.

Using metalanguage control contexts, `func` is as follows:

```
let func formals ss xi =
  let rho = List.rev formals in
  let b   = body ss rho xi in
  let n   = local_slots_needed rho ss in
  fun actuals ->
    let sigma = Array.make n Nil in
    setlocals sigma 0 actuals;
    try (b sigma; Nil) with Returnx v -> v
```

Function `local_slots_needed` walks the abstract syntax to compute the maximum number of formal parameters and local variables in scope at any given point; it is 10 lines of OCaml.

We have implemented compilation steps for the METACON family, which represents object-language control contexts using the control contexts of the metalanguage, and for the FUNCON family, which uses first-class functions.

For the DATACON family, we were less ambitious. A compilation step for DATACON converts a syntactic control context to a metalanguage function, and given that we can create metalanguage functions directly, without a syntactic intermediary, we were skeptical that such a conversion would perform well. We did implement a compilation step for statements, which we compared with the FUNCON compilation step discussed in Section 5.2 below. The results confirmed our skepticism: on average, the DATACON version is $1.58 \stackrel{\times}{\div} 1.58$ times slower when names are strings ($t = 8.87$) and $1.33 \stackrel{\times}{\div} 1.43$ times slower when names are atoms ($t = 5.70$). Worse, the DATACON compilation step exhibits the same difficulty with while loops described in Section 5.2 below, and it cannot be improved by the backpatching technique described in that section.

5.1 Using control contexts from the metalanguage

Our first compiled interpreter uses metalanguage control contexts. Figure 22 shows both the compilation step and the run step for expressions; to judge the additional programming effort required, compare Figure 22 with Figure 13. The compilation step is implemented by function `exp`: it includes case analysis of the expression, recursive "compilation" of all subexpressions, and all lookups in the environment `rho`. Function `lookup` is new: given a local variable, it reports that variable's position in the `locals` array; otherwise it identifies the variable as global.

The run step comprises all code under `(fun sigma -> ...)`: it includes lookup of global variables in ξ, lookup of local variables in σ, computation of the values `v1` and `v2` of subexpressions, and application of operators. In our experiments, `getlocal` is a synonym for `Array.unsafe_get`.

The most instructive case is for `Binop`. (Function call is similar to `Binop` and is not shown.) Metalanguage functions `eval1` and `eval2` are the compiled versions of subexpressions `e1` and `e2`. The run step applies each of these functions to the given local-variable array `sigma`, then applies the operator to the results. The temporary intermediate results `v1` and `v2` are stored in let-bound variables of the metalanguage, giving this interpreter its name: METACON-LETTMP.

The structure of the `exp` code in Figure 22 is quite like the structure of the uncompiled version in Figure 13. The `body` code in Figure 23 is much less similar to its progenitor in Figure 14. To simplify the introduction of the compilation step and the extension of the environment by a `Local` statement, we split the `body` code into three pieces: `stmts` compiles a sequence of statements, extending the environment as needed; `newrho` (not shown) computes the environment that follows a statement; and `stmt` compiles an individual statement. Only the cases for `WhileDo` and `Local` are shown. Because we are using metalanguage control contexts, we can implement `WhileDo` using a metalanguage `while`. And `setlocal`, similarly to its counterpart `getlocal`, is a synonym for `Array.unsafe_set`.

The benefits of compilation depend on the cost of name lookup. On average, when names are strings, METACON-LETTMP runs $2.75 \stackrel{\times}{\div} 1.73$ times faster than its uncompiled counterpart, METACON-IMMRHO ($t = 16.28$). But when names are atoms, which are compared for equality in constant time, METACON-LETTMP runs only $1.33 \stackrel{\times}{\div} 1.18$ times faster than METACON-IMMRHO ($t = 12.36$). Averaged over all vertices of the string cube, METACON-LETTMP runs $1.82 \stackrel{\times}{\div} 1.69$ times faster than METACON-IMMRHO ($t = 15.40$).

```
type locals = value array
type 'a cont = locals -> 'a
type 'a compiled_stmt = value cont -> value cont
type 'a kont = value  -> 'a cont
type 'a compiled_exp = 'a kont -> locals -> 'a
val exp  : exp -> env -> globals -> 'a compiled_exp
val body : stmt list -> env -> globals -> 'a compiled_stmt
```

Figure 24. FUNCON-CLOTMP type signatures

```
let exp e rho xi =
  let rec eval e = match e with
  | Lit v -> (fun k -> k v)
  | Binop (e1, op, e2) ->
      let op = binop op in
      let eval1 = eval e1 in
      let eval2 = eval e2 in
      (fun k ->
        eval1 (fun v1 ->
          eval2 (fun v2 ->
            k (op v1 v2))))
in eval e
```

Figure 25. FUNCON-CLOTMP compile & run steps (expressions)

```
let body ss rho xi =
  let rec stmts ss rho theta = match ss with
  | [] -> theta
  | s :: ss ->
      let eval e = exp e rho xi in
      match s with
      | If (e, t, f) ->
          let theta = stmts ss rho theta in
          let t = stmts t rho theta in
          let f = stmts f rho theta in
          eval e (fun v -> if not_nil v then t else f)
      | Return e -> eval e (fun v sigma -> v)
      | WhileDo (e, b) ->
          let theta = stmts ss rho theta in
          let eval_e = eval e in
          fix (fun theta' ->
            let b = stmts b rho theta' in
            eval_e (fun v ->
              if not_nil v then b else theta))
      (* other cases not shown *)
  in stmts ss rho
```

Figure 26. FUNCON-CLOTMP compile & run steps (statements)

Function `stmts` in Figure 23 allocates a closure for every statement in a sequence. As an alternative, we implemented a function that instead allocates a list of compiled statements, plus a single closure for the whole sequence. This alternative implementation of `stmts` is $1\% \stackrel{\times}{\div} 1.04$ slower ($t = 4.72$).

5.2 Using control contexts represented by functions

Continuation-passing style, as used in Figure 19, *already* maps syntax and an environment to a function from state to value. You might hope that all we need to do is to change the state type from `unit` to `value array` and we would have our compilation and run steps. For statements, this tactic almost works, but for expressions, it does not work at all. The problem with expressions is that an expression continuation expects a *value*, and values are not available until run time. In code like the `Binop` case in Figure 19, we need to lift the call to `eval e2` out of the continuation; otherwise `eval e2` will not be called until `v1` is available, in the run step. We therefore have to change `exp` so that it does not receive a continuation during the compilation step. As compiler writers, we find this restriction odd, because we are accustomed to thinking of a continuation as a

```
let body ss rho xi =
 let rec stmts ss rho theta = match ss with
 | [] -> theta
 | s :: ss ->
     let eval e = exp e rho xi in
     match s with
     | WhileDo (e, b) ->
         let theta'_ref = ref (fun _ -> assert false) in
         let compiled_b =
           stmts b rho (fun sigma -> !theta'_ref sigma) in
         let compiled_e = eval e in
         let theta      = stmts ss rho theta in
         let theta' =
           compiled_e (fun v sigma ->
             if not_nil v
             then compiled_b sigma
             else theta sigma) in
         theta'_ref := theta'; (* backpatch *)
         theta'
     (* other cases not shown *)
 in stmts ss rho
```

Figure 27. Loop compilation by backpatching (Control contexts are represented by functions)

compile-time entity, but as shown in Figures 24 to 26, the implementation is not bad.

Figure 24 shows the type signatures, and Figure 25 shows some of the code used to compile expressions. The key idea is that the expression continuation `k` is not supplied until the run step. As usual, the `Binop` case illustrates the interesting points: to analyze the abstract syntax, we make recursive calls to `op` and to `eval`; the run step receives a continuation and then proceeds very much as in the uncompiled Figure 19. The code for variables is roughly similar to Figure 22 and is omitted.

The compilation of statements is simpler, because we *can* supply statement continuations in the compilation step. Figure 26 shows the code, in which we have taken one liberty: because κ_r is always $\lambda v.\lambda\sigma.v$, we have inlined κ_r, and we do not pass it as a parameter. In the cases for `If` and `Return`, it is easy to visit all the abstract syntax in the compilation step. The same is not true of the `WhileDo` loop: the fixed-point operator delays the recursive call `stmts rho b theta'`. As a result, the body of a while loop is recompiled on each run.

When names are represented as strings, which is where the uncompiled counterpart FUNCON-IMMRHO performs less well, FUNCON-CLOTMP runs $1\% \stackrel{\times}{\div} 1.28$ faster than FUNCON-IMMRHO ($t = 0.14$). But when names are represented as atoms, FUNCON-CLOTMP runs $12\% \stackrel{\times}{\div} 1.29$ slower than FUNCON-IMMRHO ($t = 1.63$). Neither of these differences is statistically significant. Because FUNCON-CLOTMP does not compile the bodies of loops, adding a compilation step does not improve performance. We address the problem by eliminating the fixed-point operator.

In a call-by-value metalanguage, a fixed-point operator requires an explicit lambda-abstraction, which prevents the compilation step from reaching the bodies of loops. In the following definition, adapted from Figure 8, we replace the explicit fixed-point operator with a recursion equation for θ':

$$\mathcal{S}[\![\texttt{while}\ e\ \texttt{do}\ ss]\!]\theta = \theta'$$
$$\text{where } \theta' = \mathcal{E}[\![e]\!](\lambda v.\mathbf{if}\ v \neq nil\ \mathbf{then}\ \vec{\mathcal{S}}[\![ss]\!]\theta'\ \mathbf{else}\ \theta)$$

A compiler writer would solve this equation by *backpatching*: emit a fresh assembly-language label L, compile the loop under the assumption that L represents θ', and emit the compiled code immediately after L. In our compilation step, we allocate a fresh mutable reference cell, compute θ' under the assumption that the reference cell contains θ', and finally update the cell to hold θ'. Figure 27 shows the code. The resulting interpreter, FUNCON-BACKPATCH, runs $1.92 \stackrel{\times}{\div} 1.62$ times faster than the uncompiled FUNCON-IMMRHO ($t = 18.36$), and it is just $1.83 \stackrel{\times}{\div} 1.50$ times slower than Lua 2.5 ($t = 20.18$). It is $10\% \stackrel{\times}{\div} 1.07$ slower than our other compiled interpreter, METACON-LETTMP ($t = 18.29$).

6. A stack machine and a register machine

In our METACON and FUNCON interpreters, each intermediate result from `eval` is let-bound or lambda-bound. We have also emulated classic bytecode interpreters, which put intermediate results elsewhere:

- On a stack (interpreter FUNCON-STKTMP)
- In the local-variable array (interpreter FUNCON-ARRTMP-CLOIDX)

These choices correspond roughly to the "stack machine" and "register machine" of traditional bytecode interpreters. Unfortunately, in a safe functional language, simple imitations of stack machines and register machines do not tell us much. For this reason, and to make more space for presentation of other experimental results, we say little about these two families of interpreters.

Interpreter FUNCON-STKTMP uses an explicit stack of intermediate values, plus the backpatched loops from Figure 27. But the stack is implemented using heap-allocated cons cells, and it makes little difference to performance: on average, FUNCON-STKTMP is $4\% \stackrel{\times}{\div} 1.07$ slower than FUNCON-BACKPATCH ($t = 7.57$).

We also implemented an alternative that stores intermediate results in the array that holds the values of local variables. No dynamic allocation is required, which sounds promising. But using the local-variable array is 13% to 21% slower than using a heap-allocated stack (geometric standard deviation at most 1.2; $t \geq 2.18$). Profiling shows hot spots in functions `caml_page_table_lookup` and `caml_modify`. Function `caml_modify` implements a write barrier, and we conclude that the cost of getting local-variable assignments past the write barrier outweighs any gains we might have made by avoiding allocation.

7. Hybrid interpreters

As noted in Section 4, there is no reason to believe that a single representation of control contexts will perform best for both expressions and statements—especially since statements have a control operator and expressions do not. In this section, we report on the performance of some "hybrid" interpreters.

We built an uncompiled META/DATA-IMMRHO hybrid using the METACON expression code in Figure 13 and statement code similar to the DATACON code in Figure 17. The average performance of this interpreter is $1\% \stackrel{\times}{\div} 1.05$ slower of that of the non-hybrid METACON-IMMRHO interpreter ($t = 2.16$), and on average, using compact contexts makes no statistically significant difference ($t = 1.89$). We also built a hybrid META/DATA-MUTRHO with a mutable ρ. On average, it runs $1\% \stackrel{\times}{\div} 1.05$ faster than METACON-MUTRHO ($t = 2.25$). Finally, we built an uncompiled META/FUN-IMMRHO hybrid. On average it is $4\% \stackrel{\times}{\div} 1.08$ slower than the METACON-IMMRHO interpreter ($t = 6.35$) but $12\% \stackrel{\times}{\div} 1.09$ faster than the FUNCON-IMMRHO interpreter ($t = 17.31$).

To hybridize interpreters that have a compilation step, we started with the expression code from METACON-LETTMP in Figure 22. This code is a clear winner: the run step does no case analysis and allocates no data structures or closures, and intermediate values are fetched directly from closures at known offsets. We used this code together with a statement interpreter that uses first-class

functions as control contexts, backpatching loop compilation (Figure 27), and no metalanguage exceptions. The resulting hybrid interpreter, META/FUN-LETTMP, reminds us of threaded code (Bell 1973). And unlike the code in Figure 23, the hybrid interpreter uses a different indirect-branch instruction to return from each syntactic form, giving hardware branch prediction an opportunity to do a better job (Ertl and Gregg 2003). On average, this hybrid performs as well as the simpler METACON-LETTMP interpreter, only $1.67 \stackrel{\times}{\div} 1.54$ times slower than Lua 2.5 ($t = 15.99$). But on the top-performing ASN vertex of the string cube, the hybrid is $4\% \stackrel{\times}{\div} 1.12$ slower than METACON-LETTMP on average, albeit with no statistical significance ($t = 1.19$).

We did one more experiment with hybrid interpreters. Because backpatching can be tricky, we tried using a metalanguage control context for the body of a `WhileDo` loop, as in Figure 23, while using first-class functions as contexts for the other statements. Using this alternative, we had to revert to an exception handler—another metalanguage control context—to implement `Return`. The experiment did not pay off: averaged over all benchmarks, the hybrid is $4\% \stackrel{\times}{\div} 1.12$ slower than META/FUN-LETTMP ($t = 4.21$). The exact numbers vary slightly for each vertex of the string cube.

8. Benchmarking details

We compiled our interpreters to native $x86$ code using OCaml version 3.10.2 with default options. We measured elapsed time on a lightly loaded AMD Phenom 9850 Black Edition CPU running at 2.5GHz with 4GB of RAM. Each benchmark run took over eight hours.

Bagley (2002) provides 15 microbenchmarks for Lua; we do not enumerate them here. Two of the benchmarks (`lists` and `hash`) trigger pathological behavior in Lua 2.5's hashing algorithm. Both benchmarks create hash tables containing over 100,000 elements. In the `lists` benchmark, the pathology appears in Lua 2.5 itself; between 35,000 and 36,000 elements, run time suddenly quadruples. Our reimplementation is unaffected, with the result that our code runs up to 20 times faster. In the `hash` benchmark, both Lua 2.5 and our representation behave anomalously, in that running time does not increase monotonically with the number of operations performed. Both are anomalous, but our reimplementation is pathological: starting around 38,000 elements, every additional 8,000 elements doubles the running time.

We investigated Lua 3.0, and we found that it uses a different hashing algorithm which exhibits neither anomalies nor pathologies. But Lua 3.0 supports a form of first-class, nested functions with closures, and we were not confident that using it would constitute a fair, apples-to-apples comparison. In the future, we hope to backport Lua 3.0's hashing algorithm to Lua 2.5, and also to reimplement it in OCaml. For the present, we have omitted the `lists` and `hash` benchmarks from all results presented in this paper.

9. Related work

The value of using atoms instead of strings is well known. The technique of interning strings has its roots in Lisp (McCarthy 1960), where *symbols*, a distinguished set of atoms, are built into the language. In the Lua community, Mascarenhas (2009) has combined interning with type inference to enable Lua programs to run quickly on Microsoft's Common Language Runtime.

Reynolds (1972) transformed a definitional interpreter to free the object language from properties (evaluation order and first-class functions) of the metalanguage. Ager et al. (2003) have applied this *functional correspondence* to interderive semantics in which control contexts are represented using metalanguage contexts (evaluators), first-class functions (*continuations*), and data (*defunctionalized continuations*). Danvy's (2006) larger research program suggests that all three families of interpreters can be interderived as well. Our work complements this work; even when implementations can be derived systematically, it is helpful to know how different implementations can be expected to perform and how their performance depends on low-level representations.

Most work on improving definitional interpreters uses sophisticated program-transformation tools or sophisticated programming languages. Among the earliest and most often used techniques is partial evaluation (Jones, Sestoft, and Søndergaard 1985). A more recent technique, type-directed partial evaluation (Danvy 1996), scales up to an imperative language with block structure, subtyping, and higher-order functions (Danvy and Vestergaard 1996).

Pašalic, Taha, and Sheard (2002) present another strategy for improving a simple definitional interpreter: they describe an interpreter with a compilation step and a run step, but the run step is *lifted* using a staged programming language. Their paper uses a statically typed object language, and it focuses on eliminating the *tags* on datatypes such as our `value` type in Figure 10. The results are achieved using dependent types. Remarkably, Carette, Kiselyov, and Shan (2009) solve the same problem without resorting to dependent types, staging, or any other fancy language extension. Their insight is to encode object-language terms using combinator functions, not algebraic data types.

Our techniques do not begin to approach the sophistication of most of the techniques described above. But in compensation, our techniques require only simple tools, languages, and encodings.

10. Discussion and conclusion

Table 28 summarizes the performance of the most interesting interpreters in the paper, as well as the effort required to create them. An interpreter's performance is the ratio of its running time to the running time of the mature bytecode interpreter for Lua 2.5. We show the ratio at the best-performing vertex of the string cube, ASN, plus an average over all vertices. Smaller ratios are better.

Programming effort is best evaluated by studying code, but we also count source lines. Table 28 shows sizes of the `exp` and `body` functions excerpted in the paper; it also shows total line counts, which include `func`.

Compared with the amount of code needed to get a complete, working interpreter, the interpreter cores are *tiny*: around 100 lines or less. The initial basis and operators of μLua take 270 lines; a lexer and parser take 580 lines. Common support shared among compilation steps takes 60 lines, and other essential code totals 160 lines. In addition, we spent about 1000 lines on replicating Lua 2.5's hashing algorithm and on experimental scaffolding.

Table 28 suggests that introducing a compilation step might double the size of a simple interpreter core. More elaborate compilation strategies take more code. From the examples, you can judge for yourself whether interpreters with compilation steps are significantly harder to understand.

In conclusion, if there are not too many control operators, interpreters based on natural semantics are both fast and easy to write. When there are control operators, a continuation semantics is probably easier to implement, but in our very limited experiments, the resulting interpreter doesn't perform as well as one based on a natural semantics. Using a compilation step, it is easy to build a virtual machine in which each instruction is just another closure, but it is not easy to make every control transfer a proper tail call. And although a compilation step doubles the speed of many inter-

Interpreter	Ratio at vertex ASN (Avg.)	Lines of code exp	body	Total
Simple interpreters				
METACON-IMMRHO	2.04 (3.04)	11^a	28	43
DATACON-IMMRHO	2.31 (3.34)	22	35	59
FUNCON-IMMRHO	2.47 (3.52)	18	29	50
Simple interpreters (mutable ρ)				
METACON-MUTRHO	3.90 (6.08)	11^b	31	45
DATACON-MUTRHO	4.01 (6.27)	22	38	62
FUNCON-MUTRHO	4.27 (6.49)	17	31	51
With compilation steps				
METACON-LETTMP	1.52 (1.67)	27^d	50	85
FUNCON-CLOTMP	2.78 (3.77)	35^g	46	89
FUNCON-BACKPATCH	1.67 (1.83)	35^g	52	95
FUNCON-STKTMP ("stack")	1.82 (1.90)	35	67^e	110
FUNCON-ARRTMP-CLOIDX ("register")	2.15 (2.25)	33	67^e	108
Simple hybrids				
META/DATA-IMMRHO	2.07 (3.06)	11^a	30	43
META/FUN-IMMRHO-COMPACT	2.06 (3.06)	11^a	33	46
Hybrids with compilation				
META/FUN-LETTMP	1.58 (1.67)	27^d	67^e	102
META/FUN-LETTMP-METAWHILE	1.61 (1.73)	27^d	54	89

Table 28. Run times and line counts of selected interpreter cores

"Ratios" are ratios of run time to that of Lua 2.5; smaller is better. Line counts with identical superscripts represent identical code; a count with a unique superscript represents a duplicate of a function in an interpreter that is not shown. "Totals" include counts for `func`, which are not shown.

preters, when names are represented as atoms the average speedup is only 23%. Finally, in a functional language, register-based and stack-based virtual machines are not such good ideas: a register-based machine is slowed by write barriers, and allocating a value stack on the metalanguage heap has no benefits.

Using different benchmarks, a different version of OCaml, or a different target architecture would change our quantitative conclusions. We hope that the future will afford opportunities to add more modern benchmarks and to evaluate similar interpreters written in Standard ML, for which several compilers are available.

Acknowledgments

We thank Julie Farago and Gregory Price for their contributions to the early development of our interpreters. And we thank the anonymous reviewers for their helpful observations, especially about programming effort.

References

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003 (August). A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19.

Doug Bagley. 2002. The great computer language shootout. Originally at `http://www.bagley.org/~doug/shootout/`, some of this material is archived at `http://web.archive.org/`.

James R. Bell. 1973 (June). Threaded code. *Communications of the ACM*, 16(6):370–372.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543.

Olivier Danvy. 1996. Type-directed partial evaluation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257.

Olivier Danvy. 2006 (October). *An Analytical Approach to Programs as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark.

Olivier Danvy and René Vestergaard. 1996. Semantics-based compiling: A case study in type-directed partial evaluation. *Lecture Notes in Computer Science*, 1140:182–197.

M. Anton Ertl and David Gregg. 2003. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25.

Matthias Felleisen. 1987. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University. The author now prefers the revised theory presented by Felleisen and Hieb (1992).

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex*. MIT Press.

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271.

Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. 1996 (June). Lua — an extensible extension language. *Software—Practice & Experience*, 26(6):635–652.

Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. 2007 (June). The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, pages 2-1–2-26.

Neil D. Jones, Peter Sestoft, and Harald Søndergaard. 1985. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications, First International Conference*, *LNCS* volume 202, pages 124–140. Springer.

Gilles Kahn. 1987. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, *LNCS* volume 247, pages 22–39. Springer-Verlag.

Peter J. Landin. 1964 (January). The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320.

Fabio Mascarenhas. 2009 (September). *Optimized Compilation of a Dynamic Language to a Managed Runtime Environment*. PhD thesis, Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil.

John McCarthy. 1960 (April). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195.

Emir Pašalic, Walid Taha, and Tim Sheard. 2002 (September). Tagless staged interpreters for typed languages. *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming,* in *SIGPLAN Notices*, 37:218–229.

Gordon D. Plotkin. 1981 (September). A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Aarhus, Denmark.

John Reynolds. 1972 (August). Definitional interpreters for higher-order programming languages. In *Proceedings of the 25^{th} ACM National Conference*, pages 717–740. ACM.

John C. Reynolds. 1998 (December). Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361.

J. Michael Spivey. 2004 (March). Fast, accurate call graph profiling. *Software—Practice & Experience*, 34:249–264.

Joseph E. Stoy. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press.

Source code and other supplementary material is available from `http://www.cs.tufts.edu/~nr/interps.html`.