The evolution of Lua, continued

Roberto Ierusalimschy[a], Luiz Henrique de Figueiredo[b], Waldemar Celes[a]

[a]*Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil*
[b]*IMPA, Rio de Janeiro, Brazil*

Abstract

Lua is a scripting language created in 1993 in Brazil. We have reported in detail on the birth of Lua and its evolution until 2007. Here, we chronicle the evolution of Lua since then. In particular, we discuss in detail the evolution of global variables, the introduction of integers, and the implementation of garbage collection and finalizers, including deterministic finalization. We also comment on some landmark social developments in the history of Lua.

Keywords: history of computing, scripting languages

1. Introduction

Lua is a scripting language created in 1993 at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. In these 30+ years, Lua has been widely used in all kinds of industrial applications and is one of the leading scripting languages in game development [1, 2].

For HOPL III (the third ACM SIGPLAN conference on the history of programming languages), we reported in detail on the birth of Lua and its evolution until 2007 [3]. In this paper, we chronicle the continued evolution of Lua since then. Our goal is to record the main decisions made in each new version and the rationale behind them. We also review them in hindsight. After a brief overview of Lua in §2, we give a summary of the main changes in the evolution of the Lua 5 series in §3 and a detailed technical discussion of some main features and their evolution in §4. We also comment on some landmark social developments in the history of Lua in §5.

2. Overview of Lua

Lua is by design a lightweight embeddable scripting language. Our implementation of Lua is a small C library that does not bloat host programs. (Like others [4, 5], we consider code size important for a commodity library like Lua [6, 7].) Since quite early in its evolution [3], Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua supports several flavors of programming, such as procedural, object-oriented, functional, and data-driven programming [8]. Moreover, Lua has a C API that allows it to be easily embedded into host programs, thus allowing host programs to be scripted and Lua scripts to leverage

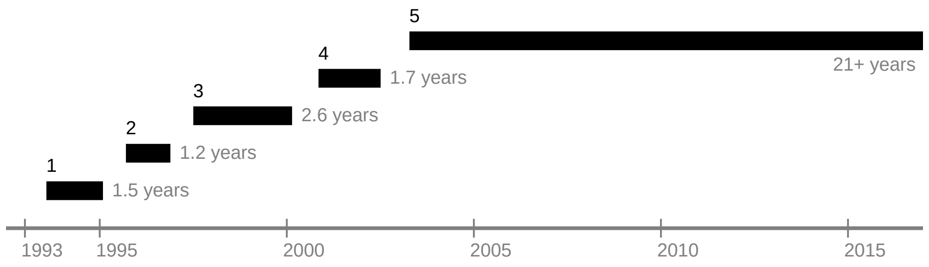

Figure 1: Life span of each Lua series: the time between the first and the last releases within a series.

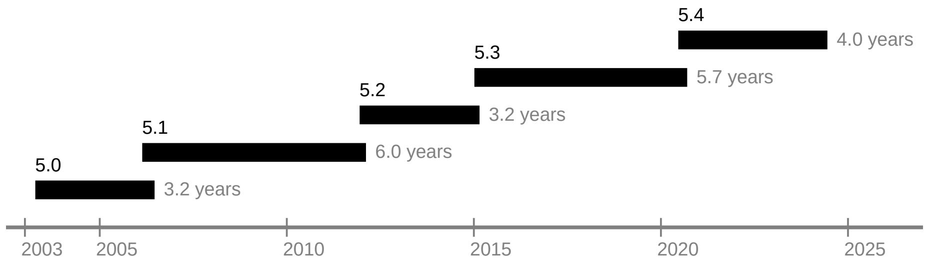

Figure 2: Life span of Lua 5 releases. The spans overlap slightly because we make a last release of the previous version shortly after a new version.

existing C libraries. This embeddability has significantly influenced the design of Lua [9, 6].

Lua offers native collaborative multithreading in the form of *coroutines* [10]. Coroutines in Lua are asymmetric, stackfull, and first-class values [11]. Support for coroutines in the language is provided by library functions. We create a coroutine with `coroutine.create`, which receives a function as the body for the new coroutine and returns a (first-class) value representing the coroutine. The function `coroutine.resume` receives a coroutine (plus optional arguments) and starts or resumes (continues) its execution. Finally, the function `coroutine.yield` yields (suspends) the execution of the coroutine calling it.

The complete definition of Lua is contained in its reference manual [12]. For a detailed introduction to Lua, see the book *Programming in Lua* [13].

3. Summary of evolution

The first versions of Lua were short-lived (see Figure 1) because Lua was changing fast, reflecting our improving understanding of the demands on a scripting language, coming from users in both PUC-Rio and elsewhere. This initial evolution of Lua is discussed at length in the HOPL paper [3]. Whenever context is needed here, we briefly recall the parts of Lua's past that are relevant to present discussion.

The Lua 5 series is by far the longest series of Lua releases; it reflects the path to both maturity and stability of the language. The series started with Lua 5.0, released in

April 2003, and is still current after 20+ years: Lua 5.4.7 was released in June 2024 and work has begun on Lua 5.5.

Lua 5 brought modern features to Lua: collaborative multithreading via Lua coroutines [10], full lexical scoping, and metatables for simpler extensible semantics. Starting with Lua 5, we adopted the liberal MIT license, which allowed Lua to be more widely used. Lua 5.1 was released in February 2006 and lasted until Lua 5.1.5, released in February 2012. It is the longest-lived version of Lua (see Figure 2); its life span was further extended by the decision of LuaJIT to stay mostly on Lua 5.1 (LuaJIT is a just-in-time compiler for Lua 5.1; see §5). Lua 5.0 and 5.1 are discussed in detail in the HOPL paper [3]. In this section, we summarize the changes in later versions in the Lua 5 series. Details on the evolution of some selected features appear in §4.

3.1. Numbering scheme

A word is in order about the scheme used for numbering versions of Lua, because it superficially resembles the popular semantic versioning scheme but is not the same, especially regarding compatibility. The releases of Lua are numbered $x.y.z$, where x is the series, $x.y$ is the version, and z is the release.

Different releases of the same version correspond to bug fixes; they have the same reference manual, the same virtual machine, and are ABI compatible.

Different versions are really different. The API is likely to be a little different (but with compatibility switches), and there is no ABI compatibility: applications that embed Lua and C libraries for Lua must be recompiled. The virtual machine is also very likely to be different in a new version: Lua programs that have been precompiled for one version will not load in a different version.

Changes in the series signal major changes in the character of the language. They happened in the early stages of the evolution of Lua. In particular, to support multiple states, the move from Lua 3 to Lua 4 changed all functions in the API to receive a Lua state as an extra parameter. From Lua 4 to Lua 5, most functions in the standard libraries were moved into modules (tables): So, `sin` became `math.sin`, `openfile` became `io.open`, `tinsert` became `table.insert`, etc.

3.2. Lua 5.2

Lua 5.2 was released in December 2011 and lasted until Lua 5.2.4, released in March 2015. Among its main new features were a new lexical scheme for globals, ephemeron tables [14], a goto statement, finalizers for tables, and yieldable C calls (suspension of coroutines while running a C function).

Globals. Since Lua 4.0, the global environment, which keeps all global variables, is stored in an ordinary Lua table, called the table of globals. This greatly simplifies the implementation of Lua and allows easy introspection using standard table mechanisms. Along with full lexical scoping, Lua 5.0 introduced environment tables that can be attached to Lua functions; the environment table of a Lua function is where global names in the function are resolved at run time. Lua 5.2 brought a new lexical scheme for globals: All variable names not declared as locals — that is, “global” names — are now resolved explicitly as fields in a table named `_ENV`; that is, any free name `foo` is converted by the compiler to `_ENV.foo`. At the same time, each chunk (the unit of compilation in Lua) is compiled in an environment with an implicit declaration of a local variable `_ENV`. By initializing this variable with different values, we can have different semantics for "global" variables. For instance, by initializing `_ENV` with a predefined global table, we have conventional global variables. On the other hand, making `_ENV=nil` is enough to block the access to any global variable. This mechanism is both simpler and more modular than the previous one based on environment tables, while keeping the flexibility. For a detailed discussion on globals, including more examples of using `_ENV`, see §4.1.

Ephemeron tables. Lua 5.0 introduced support for weak tables. A weak table is a table whose elements are weak references. Weak references are ignored by the garbage collector: If the only references to an object are weak references, then the garbage collector will collect that object. A weak table can have weak keys, weak values, or both. A table with weak keys allows the collection of its keys, but prevents the collection of its values. A table with weak values allows the collection of its values, but prevents the collection of its keys. A table with both weak keys and weak values allows the collection of both keys and values. In any case, whenever the key or the value is collected, the whole pair is removed from the table.

Lua 5.2 introduced the notion of an *ephemeron table* [14], a special kind of table with weak keys. In Lua 5.1, a strong value in a (reachable) table with weak keys is always considered reachable, independently of the key. In a (reachable) ephemeron table, a value is considered reachable only if its key is reachable. In particular, if the only reference to a key comes through its value, and the only reference to the value comes from the table itself, then the pair is removed and both the key and the value are collected. For further discussion on ephemeron tables, see §4.5.

Goto statement. A `continue` statement and other loop controls were a recurring demand from users. We were reluctant to add a `continue` statement because we thought some variant of `break` could suffice (we still do). In Lua 5.2, we opted to introduce a `goto` statement instead because it is more general. Once the focus of controversy [15, 16], the proper use of `goto` statements has long since been well understood. Nevertheless, the semantics of `goto` in Lua are delicate, as in all other programming languages that support it, because there are issues of which jumps are valid.

The syntax is `goto` *label*. The syntax for a label is `::`*label*`::`. The usual syntax *label*`:` would conflict with method calls; moreover, we thought that labels deserved a rococo syntax to highlight their presence. A label is visible in the entire block where it is defined, except inside nested functions. A `goto` may jump to any visible label as long as it does not enter into the scope of a local variable. The last rule ensures that all local variables are properly initialized. (C++ has a similar rule, that a `goto` cannot bypass a variable initialization. In Lua, a variable without an explicit initialization is implicitly initialized with `nil`, and so any variable declaration is a variable initialization.)

We decided to make a label a statement and took the opportunity to clarify the role of the semicolon as an optional statement separator, not terminator. For this, Lua 5.2 introduced the empty statement, which allows the programmer to separate statements with semicolons, start a block with a semicolon, or write multiple semicolons in sequence. The empty statement also helps to resolve ambiguity in statements starting with parentheses. As an example, consider the following fragment:

```
-- this is not an assignment followed by a function call!
a = b
(foo or goo)(a)
```

As the syntax in Lua is free format, that code is equivalent to this fragment:

```
a = b(foo or goo)(a)
```

We can fix this issue with an extra semicolon (an empty statement):

```
a = b
;(foo or goo)(a)
```

As the unusual feature in this example is the statement starting with a parentheses, it is customary in Lua to place the semicolon there.

The introduction of `goto` in Lua 5.2 was met with virtually no reaction from the community; there are still requests for a `continue` statement.

Finalizers for tables. Between Lua 4.0 and 5.2, Lua supported finalizers only for userdata, so that C libraries could release resources during garbage collection. Lua 5.2 reintroduced support for finalizers for tables, another recurring demand from users. Finalizers for tables were removed in Lua 4.0 due to performance issues: Before the sweep phase, the collector had to traverse all objects looking for dead objects with finalizers, so that they could be resurrected. In Lua 5.2, we added in the garbage collector a list of objects marked for finalization, which solved the problem. For a detailed discussion about finalizers in Lua, see §4.5.

Yieldable C calls. The implementation of coroutines in Lua 5.0 used a *stackless* interpreter, where calls in Lua do not imply calls in the underlying C code of the interpreter. The call stack is reified in the Lua stack, so that the interpreter can have multiple call stacks, one for each coroutine. However, calls to C functions still need the actual C stack. So, coroutines in Lua 5.0 and 5.1 cannot yield while there is a C function in the call stack (that is, while running a C function, including functions called by it).

In 2005, Mike Pall (the developer of LuaJIT) published a patch that allowed a restricted form of yields in C calls [17]. In particular, the patch allowed yields from inside metamethods and protected calls, the mechanism used by Lua for exception handling. The patch came during the development of Lua 5.1, but as we were already close to the release of Lua 5.1 alpha, it had to wait for the next version. Lua 5.2 came with our first implementation of yieldable C calls, strongly based on Mike Pall's patch. In Lua 5.3 we refined that implementation with the concept of *continuation functions*. See §4.3 for the details about the evolution of this feature.

3.3. Lua 5.3

Lua 5.3 was released in January 2015 and lasted until Lua 5.3.6, released in September 2020. It is the second longest-lived version of Lua to date (see Figure 2). Among its main new features were the introduction of integers and bitwise operators, and a basic UTF-8 library.

Integers. The main new feature in Lua 5.3 was the introduction of integers and consequently of bitwise operators and their corresponding metamethods. Until Lua 5.2, Lua had only one kind of number, implemented by default as double-precision floating-point numbers. Integers were not really needed because double-precision represents any 53-bit integer without loss. Indeed, Lua could boast support for such large integers when 32-bit integers were the norm. With the rising ubiquity of 64-bit platforms, this argument no longer applied. Indeed, by that time several C libraries were using 64-bit integers for high-precision timestamps, handles, etc. Lua programs were having to jump through hoops to use 64-bit integers. Therefore, we decided to introduce integers in Lua as an explicit subtype of numbers. The introduction of integers may seem a minor change, but it had far-reaching consequences. We discuss several aspects of this change in §4.2.

Unicode. Lua is entirely agnostic about strings: It makes no assumptions about their contents; they are just sequences of bytes. In particular, Lua does not know or care whether a string contains text in some encoding. Accordingly, the standard string library in Lua is based on bytes and naturally favors text manipulation of strings using single-byte encodings, like ISO-8859-1. Fortunately, plain string matching works correctly regardless of encodings. However, general pattern matching does not work in multi-byte encodings, because it relies on characters, not bytes: Patterns need to identify classes of characters like whitespace, punctuation, digits, letters, etc.

In response to the rising popularity of Unicode, Lua 5.3 also brought a library for basic manipulation of UTF-8 strings, UTF-8 being the most popular encoding for Unicode. Because Unicode tables are huge (much larger than the entire code of our Lua implementation), we purposely restricted this library to the manipulation of characters encoded in UTF-8; the library provides no support for Unicode other than the handling of that encoding. Operations that need the meaning of a character, such as character classification, are outside the scope of the library. In particular, it does not offer pattern matching.

The `utf8` library seems to have fulfilled its mission of granting access to individual characters in UTF-8 strings. Third-party libraries for pattern matching on UTF-8 strings exist now, but they are not small. Sophisticated pattern matching and parsing of UTF-8 and other encodings can be done elegantly in Lua using LPEG [18].

3.4. Lua 5.4

Lua 5.4 was released in June 2020. The current release as of January 2025 is Lua 5.4.7, released in June 2024. Its main new features are a new generational mode for garbage collection and to-be-closed variables for deterministic finalization (see §4.5).

Generational garbage collection. A generational collector assumes that most objects die young. Consequently, its regular cycle traverses only young objects, those that were created recently. While this behavior can reduce the time used by the collector, it can also increase memory usage, since old dead objects may accumulate. To mitigate this accumulation of garbage, the generational collector performs a full collection from time to time. Lua 5.2 introduced generational garbage collection as an experimental feature, but it did not work well: The strict binary classification of objects as old and young is too coarse. Generational garbage collection was removed in Lua 5.3 but it was reinstated as the default in Lua 5.4 after refining the classification of objects according to age. For a detailed discussion of garbage collection in Lua, see §4.4.

To-be-closed variables. To-be-closed variables are somewhat equivalent to `finally` blocks in Java and Python. They provide the key mechanism of *deterministic finalization* for Lua, which is discussed in detail in §4.5.

4. Details of evolution

In this section, we discuss in detail the evolution of the new features of Lua since 5.1 that involve more technical considerations. Namely, here we will discuss globals, integers, yieldable C calls, garbage collection, and finalizers, which also includes deterministic finalization.

4.1. Globals

Lua has always had global variables. They fit Lua's origins as a configuration language and they simplify the communication with the host program. Nevertheless, the notion of global variable and its implementation in Lua has evolved significantly.

The global environment. Until Lua 4, we used private data structures inside the interpreter to represent the *global environment*, which stores all global variables. These data structures have evolved from simple fixed-sized arrays to linked lists to binary trees to hash tables. Their privacy made it easy for us to change their representation but it also imposed a burden on Lua, because we wanted to support reflection for the global environment. Lua 1 already provided basic reflective facilities, including a special function for the traversal of the global environment that mimics the traversal of Lua tables. This trend to make the global environment feel like a Lua table continued as Lua evolved. Lua 2 added special functions to query and change global variables with names computed at run time; with this change, a variable "name" could be any string. Lua 3 introduced iterator functions for traversing Lua tables and the global environment. Finally, following a suggestion by John Belmonte, Lua 4 used an ordinary Lua table to store its global variables. This change has greatly simplified the implementation of the interpreter and has also allowed us to remove all special functions that supported reflection for the global environment; reflection now comes for free from the existing features of regular tables. Moreover, extensible semantics can be easily applied to the global environment without requiring special support since the table of globals, like any Lua table, can be given a metatable. In particular, extensible semantics can be used for tracking access to global variables, making the global environment read-only, implementing sandboxing, etc. After explaining the new environment mechanism for Lua 5.2, we will show how it supports these and other uses.

Environment tables. Lua 5.0 introduced environment tables that can be attached to Lua functions; these are the tables where global names in a function are resolved at run time. Lua 5.1 extended environment tables to C functions, userdata, and threads, thus effectively replacing the notion of global environment. Lua provided special functions (`setfenv` and `getfenv`) to support these notions. However, these functions were at once too powerful and not powerful enough. Indeed, any code that can call a function can also change its environment; all that `setfenv` needs is a reference to the function. That ease encourages all kinds of tricks that stymie modularity. At the same time, the all-too-common case where a function wants to change its own environment was difficult to code, because there is no standard way for a function to refer to itself.

A lexical scheme. The lexical scheme for globals introduced in Lua 5.2 replaced the ad hoc manipulation of environment tables attached to Lua functions with a transparent, lexically clear construct; accordingly, the special functions `setfenv` and `getfenv` were removed. In this new scheme, Lua does not have global variables, although it goes "to great lengths to pretend it has" [13]. The compiler converts any free variable name `var` to `_ENV.var`. All Lua chunks are anonymous functions that have an implicit *upvalue* (a variable that is local to an enclosing block) named `_ENV`. The initial value of this upvalue is the global environment, a regular table. So, the expression `_ENV.var` refers to the entry indexed by the string `"var"` in the global environment; that entry is what we call the global `var`. The overall behavior is that of conventional global variables, thus maintaining the illusion.

The trick of this scheme is that there are no tricks. The translation of `var` to `_ENV.var` is literal. Except for the fact that every chunk has an upvalue with the name `_ENV`, there is nothing special about this name; it is bound following the regular rules for lexical scoping in Lua. Typically, `_ENV` means the implicit upvalue, but the compiler will use any `_ENV` that is in scope. In particular, `_ENV` can be a local variable or a function parameter. Moreover, the program can assign any value to these variables. Thus, the value of `_ENV` can be changed at run time in a transparent, lexically clear construct: Just assign `_ENV` or declare a new variable with that name. The somewhat awkward name `_ENV` is meant as a reminder that this kind of manipulation is not to be taken lightly.

Despite all this flexibility, no piece of code outside a function can change its environment unless the function itself provides some support for that. This constraint enhances the role of functions as the main mechanism for modularity in Lua [6].

It is worth emphasizing that the implementation of the lexical scheme for globals is mostly limited to the compiler. Outside the compiler, it requires only a couple of details:

- The registry keeps a table, which is used as the standard table of globals.
- After a new chunk is compiled, the interpreter sets this table of globals as the initial value of the first upvalue of the chunk (the implicit `_ENV` variable).

As a final touch for convenience, the error-reporting mechanism "corrects" references to `_ENV` fields when creating error messages:

```
$ lua
> a + 1
```

```
--> attempt to perform arithmetic on a nil value (global 'a')
> _ENV.a + 1
--> attempt to perform arithmetic on a nil value (global 'a')
> t={}; print(t.a + 1)
--> attempt to perform arithmetic on a nil value (field 'a')
```

Some examples. Here are some examples of uses of `_ENV` [13, chapter 22]:

- Forbidding access to global variables: Setting `_ENV=nil` invalidates direct access to global variables in the rest of the chunk. This is useful to control what global variables are being used, for both reading and writing. Here is a typical use:

  ```
  local sin = math.sin    -- _ENV.math.sin
  local cos = math.cos    -- _ENV.math.cos
  _ENV = nil
  -- no more access to globals;
  -- only 'sin' and 'cos' are available
  ```

- Tracking access to global variables: Giving appropriate metamethods to `_ENV` allow us to track all accesses to global variables in the rest of the chunk. The following code does the trick:

  ```
  local _ENV = setmetatable({},
    {__index = function (_, key)
                 print("read", key)
                 return _ENV[key]
               end,
     __newindex = function (_, key, val)
                    print("write", key, val)
                    _ENV[key] = val
                  end})
  ```

  We create a new local variable `_ENV` with an empty table and appropriate metamethods. Because the table is empty, any read or write to this table triggers a corresponding metamethod (`__index` for a read, `__newindex` for a write). Both metamethods print a log message and then perform the desired operation in the original environment. (A subtle point is that the new `_ENV` is visible only after its declaration, so the `_ENV` used in its initialization bounds to the original `_ENV`.) This technique is known as *proxying*; it can be applied to any table, not just `_ENV`.

- Read-only environment: To avoid some code changing global variables, we can use proxying, except that the read metamethod just repeats the access in the original environment and the write metamethod raises an error:

  ```
  local _ENV = setmetatable({},
    {__index = _ENV,
     __newindex = function (_, key, val)
                    error("invalid write")
                  end})
  ```

- Sandboxing: To run some (probably untrusted) code in a controlled environment, populate a new table with trusted functions and assign it to `_ENV`. The remainder of the chunk will be evaluated in this trusted environment. Alternatively, assign to `_ENV` an empty table that inherits from the original `_ENV`:

  ```
  _ENV = setmetatable({}, {__index = _ENV})
  ```

  This allows code to read existing global variables while protecting them from writing: All global assignments will go to the empty table.
- Flexible functions: A function having `_ENV` as a parameter can be evaluated in multiple environments that are given dynamically, as an argument. For example, `function foo (_ENV,...)`.

4.2. Integers

The very first version of Lua, released in 1993, had only single-precision floating-point numbers. JavaScript, from 1995, had (and still has) only double-precision floating-point numbers. In 1998, Lua 3.1 changed the numeric type to double-precision numbers. Curiously, the main reason for that change was not directly related to float precision. Single-precision numbers have a mantissa of 24 bits and therefore can represent exact integers only up to 2^{24}, approximately 16 million. For systems that measure time like Unix, counting seconds from some epoch, 24 bits last less than one year. Signed 32-bit integers last 68 years. The mantissa of double-precision numbers has 53 bits, which can count individual seconds for over 285 million years.

Double-precision numbers offer several advantages. They have a very well-defined behavior, dictated by the ubiquitous IEEE 754 standard [19]. Most conventional platforms have hardware support for them. Their 53-bit mantissas allow them to represent integers precisely up to more than 10^{15}. The language itself becomes simpler by having a single unifying numeric type.

However, double-precision numbers also have their drawbacks. First and foremost, they cannot represent 64-bit integers fully. While few programs need 64-bit integers for counting, several modern algorithms, such as cryptography, do need them, Also, 64-bit integers are used as handles in APIs, for instance in Windows. A second drawback regards performance in restricted hardware. Lua is quite popular in embedded systems [1], which typically lack hardware support for double-precision arithmetic; moreover, their restricted memory would benefit from a smaller representation of numbers in Lua.

A third, often neglected drawback is that integers were already present in the language, but hidden as a second-class type. Several functions in the API have integer parameters, such as the indices in the string library. Nevertheless, the language did not specify how a Lua number was converted to those integer values. As an example, what should be the result of the following expression?

```
string.sub("hello", -2.1, 3.9)
```

Is this an error or should Lua silently convert indices to integers? If so, should it truncate or round?

A particularly problematic instance of integers as second-class values regards bitwise operations. For a long time, Lua did not have bitwise operations exactly because of the lack of a proper integer type. How should bitwise operations behave with floating-point numbers? Should they operate on 53 bits? Should they operate on signed integers? As a stopgap, Lua 5.2 introduced bitwise operations as a library, to avoid a full commitment of the language itself to any specific second-class integer representation. But a library falls short for such common operations present in most other programming languages. Moreover, it is tedious to convert existing code in other languages that use bitwise operators into function calls.

A final drawback comes from the implementation of Lua. Since version 5.0, Lua optimizes the representation of arrays (tables with consecutive integer keys) by using an actual array to store their values, thus avoiding storing the keys [20]. To index these internal arrays, the interpreter has to convert any index from a float to an integer, and that operation is somewhat slow in some architectures.

For all those reasons, we decided to add an integer numeric type to Lua, starting in version 5.3, released in January 2015. We report below the main design decisions in this change to the language, how those decisions evolved, and how the new feature worked in real life.

Alternatives. Before settling on the current implementation, we considered several alternatives to solve the drawbacks of a single double-precision numeric type, mainly using long double, using integers internally, and making integers a new type.

Long double. The same way we changed from single precision to double precision when we needed 32-bit integers, we could repeat the trick by changing the Lua numeric type from double precision to quadruple precision, whose mantissa has 113 bits [19]. That solution keeps the simplicity of a single well-defined numeric type while adding support for 64-bit integers. However, it does not solve the other issues that motivated integers in the first place. It does not address the problem of integers being a second-class type in the language. It does not address the performance issue of converting floats to integers when indexing arrays. It worsens the support of Lua for restricted hardware. A final hindrance is that ISO C does not support quadruple-precision floats. The C standard offers a type `long double`, which in the x86-64 architecture is typically an 80-bit float, with a 64-bit mantissa. (Due to alignment, it may occupy 16 bytes in memory.) However, the C standard allows this type to have the same characteristics of a regular `double`.

Integers as an "implementation detail". A large part of the design space for adding integers to Lua revolves around how programmers see this new type. One approach is to make it as invisible as possible. Lua would keep one single numeric type, but internally it could use a floating-point representation or an integer representation, depending on a set of conditions. Ideally, Lua would select, for each value, the format that can represent that value more accurately. However, the correct implementation of this semantics is prohibitively expensive, as it cannot use the operations implemented by the hardware. As an example, consider the product `(2^62 + 2) * 0.5`. The first operand has an exact representation as a 64-bit integer (but not as a double); the second operand has an exact representation as a double (but not as an integer); the result has an exact representation as a 64-bit integer. However, we cannot perform that multiplication either as an integer multiplication or as a float multiplication.

Integers as a new type. Another approach for providing 64-bit values to Lua would be to add a new type corresponding to integers. That solution keeps the uniformity of IEEE 754 for floating-point values and solves most of the problems listed above. Within this broad approach, there are still several smaller decisions to be made, especially concerning the relation between floats and integers: mixed arithmetic operations, conversions and casts between them, etc. However, independent of these smaller decisions, this approach has a major drawback: lack of compatibility. With this change, the simple check `type(x) == "number"` would fail for integer values. Because Lua did not have integers, programmers did not distinguish between `0` and `0.0`. Suddenly, any code that checks the type of a value would distinguish `0` from `0.0`.

Main design decisions. After careful consideration of those several alternatives, it became clear to us that the best way to reconcile compatibility with all the improvements expected from an integer type was to implement integers as a subtype of numbers. The type *number* would comprise two subtypes: *float* and *integer*. (The term *float* here refers to any floating-point representation, regardless of its size.) All operations would either accept both subtypes or try to convert one to the other. In particular, an integer value can always be converted to a float, perhaps losing precision. In the other direction, there were some choices to be made. For instance, a conversion from float to integer could entail some kind of rounding. After some consideration, we opted for no implicit rounding: If a float does not have an exact representation as an integer, the conversion fails. We thought that, more often than not, a non-integer value being used where an integer is expected (e.g., as an index into a string or in a bitwise operation) hints at some bug in the program. When needed, the programmer should explicitly round the value appropriately.

We adopted the following principles to guide other decisions regarding these two subtypes:

- Whenever possible, the language should avoid differences between the two subtypes. A change between `0` and `0.0` should have a minimal effect on a program.
- At the same time, the language should have explicit and clear rules regarding how each subtype behaves.

These two principles are summarized in the following rule: “The programmer may choose to mostly ignore the difference between integers and floats or to assume complete control over the representation of each number” [12].

Arithmetic operations. We started by borrowing the semantics that many languages adopt for arithmetic operations: Integer operands give an integer result; mixed or float operands give a float result.

For addition, multiplication, and subtraction, this rule matches our first principle: Because the integers are closed under these operations, the results do not depend on the subtype of the operands, except in case of overflows.

```
> 5 * 1024        --> 5120
> 5.0 * 1024.0    --> 5120.0
```

The integers are closed also under the modulo operation (%), so this operation can follow the same rule.

For division, from the start we followed the lesson from Python 3 and avoided the confusion of `5.0/2` being `2.5` but `5/2` being `2`. Regular division always operates on floats and gives a float result.

```
> 5.0 / 2   --> 2.5
> 5 / 2     --> 2.5
```

A new integer division — denoted by "`//`" (borrowed from Python 3) — floors its result to an integer. Initially, we named that operator *integer division* [21], but later we renamed it to *floor division* (also borrowed from Python), due to a subtlety: That operation is not restricted to integers. Instead, it follows the general rule for arithmetic operators: Integer operands give an integer result, mixed and float operands give a float result — although the result is always an integral value.

```
> 5.0 // 2      --> 2.0
> 5 // 2        --> 2
> 2.5 // 0.5    --> 5.0
> 1e300 // 2    --> 5e+299
```

The exponentiation operation had a more convoluted evolution. We initially considered several variations, including a design where integer operands give integer results. Under this rule, most operations with a negative exponent would result in zero: `x^-y` would be equal to `1/(x^y)`, and the cast of the fraction to an integer would result in zero. This seems neither useful nor intuitive, so we ruled out this option.

For the first implementation, we settled for a rule where an integer exponentiation was performed only when both operands were integers and the exponent was nonnegative. So, `2^2` resulted in `4` (as an integer), but `2^-2` resulted in `0.25`. However, that design was changed before Lua 5.3 alpha. To keep the type rules simple, we adopted a third principle: The subtype of an operation could depend on the subtypes of its operands, but not on the values of those operands. This principle voided our implementation for exponentiation. Instead, we settled for an exponentiation that, like division, always operates on floats and gives a float result. In common uses that is hardly an issue, except for powers of two: More often than not, when we compute powers of two we are thinking about bits and exact representations.

```
> 2^60       --> 1.1529215046068e+18
> 1 << 60    --> 1152921504606846976
```

The float `2^60` can be safely coerced to the exact integer value, but we don't see that when printed. Moreover, it also can spoil other operations after it. For instance, `2^60+1` results in `2^60`, because the addition is done with floats, which do not have enough precision for the added one.

Bitwise operations. For bitwise operations, we adopted the C operations and operators, except for the exclusive or, that is denoted by ~ in Lua. (The caret in Lua already denoted exponentiation.) All bitwise operations operate only on integers: Any float operand is converted to an integer. In particular, the expression `x|0` is an easy way to coerce a number to an integer. Following the general rule for this conversion, an error is raised if the float does not have an exact integer representation.

```
> 3.0 | 0    --> 3
> 3.1 & 3    --> error: number has no integer representation
> 1e100 & 3  --> error: number has no integer representation
```

The bitwise shift operations work with any offset. Negative offsets shift in the opposite direction; offsets larger than the integer size result in zero, as all significant bits are shifted out.

```
> 8 << 1    --> 16
> 8 << -1   --> 4
> 8 >> -3   --> 64
> 1 << 65   --> 0
```

The right shift is always a logical shift; there is no arithmetic shift. (Even ISO C does not have an arithmetic shift. According to the standard, the resulting value for a right shift of a negative value is implementation-defined [22].)

Explicit rounding. Another clouded area was the functions to convert floats to integers, namely `math.floor`, `math.ceil`, and `math.modf`. At first, we adopted the rule that types do not depend on values. An option then would be to have always integer results from these functions. However, these functions are useful for float computations too, and giving only integer results would constrain the range of values that these operations could handle. So, we opted initially for the general rule "integer input implies integer output", but that was kind of useless: We end up with no functions to convert floats to integers. (Not to mention that it makes little sense to apply these functions to integer values.) In the end, we decided that these functions, being specific for float–integer conversions, could break the rule that types cannot depend on values. For them, if the result fits in an integer, it is typed as an integer; otherwise the result is a float:

```
> math.floor(5.3)     --> 5
> math.floor(1e20)    --> 1e+20
```

(For float-only computations, any integer result will be safely converted back to a float when used in any subsequent computation that requires a float.) Moreover, to give the programmer finer control over conversions, we added another function that mimics the implicit conversion that Lua does, but returns `nil` in case of errors:

```
> 100.0 | 0               --> 100
> math.tointeger(100.0)   --> 100
> 100.5 | 0               -- error!
> math.tointeger(100.5)   --> nil
```

Integer overflows. As previously mentioned, the results of arithmetic operations over floats and integers diverge when there is an overflow. For floats, the IEEE 754 standard provides specific rules for handling overflows. For integers, we considered three options: convert to float, raise an error, or wrap around.

The first option would be to convert the result of the operation to float if it overflowed the integer representation. That option keeps strict compatibility with previous versions of Lua without an integer subtype, but it has few other advantages. In particular, it breaks the rule that the type of a result should not depend on the particular values of the operands. It also adds some overhead over arithmetic operations, for checking overflows. The second option would be to raise an error in case of overflows. That would be the cleanest design. However, it adds an overhead over arithmetic operations and rules out the use of integers as unsigned values. In the end, we adopted the third option, to wrap around results. That solution has zero overhead in any two-complement architecture and an added benefit of allowing Lua integers to represent unsigned integers.

For literals, we started with that same rule: Any integer literal too large to fit into an integer would have its value reduced modulo the integer size. However, there were some complaints in the mailing list. A main issue was that it is not uncommon to write a literal without a decimal point when it does not have a fractional part, even if the value is intended to be a float. Lua 5.2 would read back the number correctly (as a float), but Lua 5.3 would read it as an integer and then wrap around the value, giving a wrong result. So, in Lua 5.3.3 we changed that rule, so that literal decimal integers that overflow are read as floats. Literals in hexadecimal, however, are still read as integers, with wrap around. A subtle aspect of this new rule is the handling of the minimum value for integers. Like most languages, Lua reads the value `-9223372036854775808` as `-(9223372036854775808)`, that is, an unary minus applied over a positive constant. The positive constant `9223372036854775808` overflows, so Lua reads it as a float, and the unary minus applied to a float results in a float:

```
>  9223372036854775808  -->  9.2233720368548e+18
> -9223372036854775808  --> -9.2233720368548e+18
```

However, as we already discussed, this value is readily converted to an integer if needed, so that behavior is not a real issue. Moreover, programmers should seldom write this constant: Lua now provides a constant `math.mininteger` with that value with the correct type. (It also provides a corresponding `math.maxinteger`.)

Order operations. Since we started considering the inclusion of integers in Lua, we considered that order operations should respect the *usual arithmetic conversions* from C, just like the arithmetic operators. After all, that is what a CPU offers us: Either we compare two integers or we compare two floats.

Lua 5.3.0, the very first version of Lua with integers, was released with that semantics for order operators. After the release, however, we realized that it need not be that way. After all, the result of a comparison, being a boolean, can always be correctly represented, no matter the types of the operands. Lua 5.3.1 changed the semantics, so that a comparison always gives the correct mathematical result, no matter the types of the operands. As an example, consider the comparison `(1 << 60) + 1 <= 2^60`. With the first semantics, as the second operand is a float, the first operand is coerced to a float. Because a double-precision float has only 53 bits of mantissa, the operand was rounded from $2^{60}+1$ to 2^{60}, and the whole expression evaluated to true. With the new semantics, the final result is false, as expected.

The current implementation uses the following algorithm for numerical comparisons. If both operands have the same type, the standard CPU operations already give the correct result. For mixed types, if the integer operand can be represented exactly as a float — that is, its absolute value is less than or equal 2^n, where n is the number of bits in the float mantissa — that operand is converted to a float and the two floats are compared. That is by far the most common case and it is reasonably efficient. Otherwise, if the floor (or the ceiling, depending on the specific comparison) of the float operand is inside the range of integers, then it is converted to an integer and the two integers are compared. (If i is an integer, then $f < i$ if and only if $\lfloor f \rfloor < i$.) This path can be slower, due to the conversion from float to integer. Otherwise, we just check the sign of the float: A number not in the range of integers either is larger or smaller than all integers.

Conversion to string. A subtle point in the introduction of subtypes of numbers regards the conversion of numbers to strings and, consequently, the way numbers are printed. With all numbers being float, `1` and `1.0` are exactly the same thing. In particular, in previous versions, numbers without a decimal part were printed without a decimal point:

```
$ lua5.2
> print(1.0)
1
```

With the introduction of subtypes, we thought it would be useful to distinguish between printed floats and integers. So, we changed the conversion to always ensure the presence of a floating point or an exponent in a numeral representing a float:

```
$ lua5.3
> print(1.0)
1.0
> print(1)
1
> print(1000000000000000)
1000000000000000
> print(1000000000000000.0)
1e+15
```

Note that all our previous examples have used the new format, so that we can easily tell floats from integers.

The documentation has always stated that the coercion from numbers to strings followed "a reasonable format" [23], without any further details. So, this change was not strictly an incompatibility. All the same, it was perhaps the main cause of compatibility issues for programmers migrating from Lua 5.2 to Lua 5.3. Many programs broke when receiving '`1.0`' while expecting '`1`'. Nevertheless, we do not regret the change. In fact, more often than not these issues revealed small problems in the program: A computation that should be performed only with integer values was being tainted by some float value.

Variants. As we already discussed, until Lua 5.2 numbers in Lua were all double-precision floating-point numbers. The source code had some configuration macros to facilitate the change of that type. According to the documentation, "[it] is easy to build Lua interpreters that use other internal representations for numbers, such as single-precision floats or long integers" [23]; however, those macros were neither officially supported nor tested. Nevertheless, some projects have used that facility. For instance, Lunatik, a framework for scripting the Linux kernel with Lua [24], compiles Lua with a sole integer type, because the CPU cannot run floating-point operations inside the Linux kernel.

Another benefit from the new integer type is that it made single-precision floats more attractive, as the integer type allows the exact representation of 32-bit values, something that was not possible having only single-precision floats. Moreover, the whole code base became cleaner, with fewer and better-defined points of conversions between floats and integers. In the end, Lua 5.3 incorporated official support for alternative numerical types. The integer type can be `int` (usually 32 bits), `long`, or `long long` (usually 64 bits), while the float type can be `float` (usually 32 bits), `double` (usually 64 bits), or `long double`. All combinations are fully tested before any new release.

A configuration with 32-bit integers plus single-precision floats is particularly interesting. The 32-bit integers solve the problem that single-precision floats cannot store 32-bit quantities. In 32-bit architectures, where pointers also have 32 bits, a generic Lua value fits in 32 bits (plus a tag). That configuration got the name of *Small Lua*. We have anecdotal evidence of it being used in embedded devices.

During the whole process of adding integers to Lua, compatibility was always a main concern. We think we did a good job on this front: We got a few complaints about the format change from '`1`' to '`1.0`', and that was mostly it. (The Lua manual never specified how numbers are formatted when coerced to strings, so that was not even an incompatibility according to the documentation.) After we arrived at a good design, it is difficult to see how things could be different, except for some details. For instance, integer and float numbers could have different types, instead of a single type `number` for both. Overall, though, compatibility did not have a big impact in the final result.

4.3. Yieldable C calls

The implementation of coroutines in Lua 5.0 used a *stackless* interpreter. (In this context, "stackless" means that the interpreter does not use the C stack for its calls.) When the program calls a Lua function, the interpreter does not perform a corresponding call in C. Instead, it mimics a real CPU: It pushes on the Lua stack (a regular data structure) the state of the interpreter in the current function and goes to execute the code of the called function. When it reaches a return instruction, it pops from the stack the state of the caller function and continues its execution.

That scheme works pretty well when a Lua function calls another Lua function, but it cannot support calls to C functions: These calls clearly need the C stack. Therefore, Lua 5.0 disallowed yields while running a C function; a runtime check raises an error in that situation.

In general, that restriction was not a big issue. Typically, C functions implement external tasks that would not yield anyway. When a C function calls back Lua, it is often for short tasks, such as comparing two values in `table.sort` or providing a substitution value in `string.gsub`.

However, in one particular case that restriction was more than an inconvenience. Lua does exception handling through *protected calls*: The function `pcall` calls any given function in *protected mode*, where it captures any error raised during the call and returns the error message. The function `pcall` is implemented in C, and so it had that same restriction. As a result, any function running in protected mode could not yield.

In 2005, Mike Pall came up with a smart scheme to solve that restriction. In a simplified view, his scheme involved three parts.

- All coroutines are interpreted within the scope of a protecting `setjmp`. This was already the case, as errors in coroutines are not propagated through resumes.
- When there is an yield while Lua is running inside a C function, the interpreter does a long jump to the respective resume. This long jump effectively erases from the C stack any C function that was active at that time.
- When the time comes to resume a C function that was erased by the long jump, the interpreter *calls the function again*. (The function pointer is in the Lua stack.) Clearly, the function itself should be prepared to be called again. For that, it could use a new API function to know whether it was being called as a new invocation or to continue a previous call that was interrupted by an yield.

In Lua 5.3, we refined that scheme. Instead of calling again the same function to continue the work after an yield, the interpreter called another function to that end. Here is an example. Suppose we have a C function `foo` that wants to yield midway its execution. Initially, its hypothetical code looks like this:

```
// doesn't work!
int foo (lua_State *L) {
  // do some stuff
  lua_yield(L, n);  // yields n values on stack
  // do some more stuff after yielding
}
```

Clearly, this code will not work correctly, because `lua_yield` does a long jump and never returns, and so the second block is never executed. To fix that, put all code to be executed after the yield in a *continuation function*. Then change `lua_yield` to `lua_yieldk`, which allows us to pass the continuation function to Lua:

```
// continuation function
int foo_cont (lua_State *L, int status, lua_KContext ctx)) {
  // do some more stuff after yielding
}

int foo (lua_State *L) {
  // do some stuff
  lua_yieldk(L, n, 0, foo_cont);
  return 0;
}
```

(We will explain the parameters of `foo_cont` shortly.) As before, `lua_yieldk` will do a long jump and never return. (The `return` statement is dead code, although the compiler does not know that. It is there only to avoid warnings.) But when the time comes to resume `foo`, the interpreter instead calls the continuation function (which was saved in the Lua stack) to finish what `foo` had to do. Besides the ubiquitous Lua state `L`, the continuation function — `foo_cont` in the example — receives two more parameters. The second parameter, `status`, is used for error handling, which is outside the scope of this short description. The third parameter, the context `ctx`, is an arbitrary value that the original function (`foo`, in our example) can pass to its continuation. It is the third argument to `lua_yieldk` (0, in our example). Lua does not use this value for anything.

A key property of coroutines in Lua is that they are *stackfull*: A coroutine can yield from inside several levels of function calls, and then resume with that same call stack. Therefore, besides a C function yielding itself, we also want it to be able to call back to Lua allowing that Lua code to yield. The scheme we use to support these indirect yields is similar to what we presented for direct yields. When a C function wants to call a Lua function, it does so by calling an API function `lua_callk`. Like `lua_yieldk`, `lua_callk` receives a continuation function. Unlike `lua_yieldk`, which always yields, `lua_callk` may or may not yield, depending on the called Lua code. If the Lua code does not yield, `lua_callk` returns normally. Otherwise, the call is interrupted, and when the time comes to resume the interrupted C function, Lua calls the given continuation function instead.

4.4. Garbage collection

Like most dynamic languages, Lua features automatic memory management in the form of garbage collection [25]. Since its first version, Lua has used a tracing collector: The collector *traces* objects that are reachable from a root set — roughly all global and local variables — and collects those that are not reachable. Over this basic concept, the implementation of garbage collection in Lua has evolved significantly as Lua became more complex. All objects in Lua are subject to garbage collection: tables, strings, functions, threads (coroutines), userdata, modules (which are actually tables), etc. Most internal structures in the interpreter also have their existence and sizes controlled by the garbage collector: stacks, the store for internalized strings, the registry (which is a table), the global environment (which is a table), etc. The only memory increase in Lua that cannot be reversed is the binding of dynamically linked libraries.

Stop the world. From its birth in 1993 until version 5.0, Lua used a *mark-and-sweep* collector. From time to time, the interpreter runs a complete garbage-collection cycle, which comprises two main phases: *mark*, where it traverses and marks all reachable objects, and *sweep*, where it traverses all objects releasing those that have not been marked. A major drawback of mark-and-sweep collectors is that a whole cycle can take some time — after all, it has to visit all objects in the memory. While the collector is running, the rest of the program cannot progress. For that reason, that kind of collector is often called a *stop-the-world* collector [25].

Incremental collection. Stop-the-world collectors are not particularly suited to soft real-time applications, such as video games; long pauses spoil the user experience.

So, in 2006, Lua 5.1 introduced an *incremental collector*. As the name implies, an incremental collector does its job incrementally, in small steps interleaved with the *mutator*. (From the point of view of the garbage collector, the rest of the program is just some nasty code that insists on changing the reachability graph while the collector tries to do its job. For that reason, the main code is often called the *mutator* [25].)

Incremental collectors solve the problem of the long pauses caused by a conventional stop-the-world collector. However, they do nothing to improve performance in general, quite the opposite. To allow the concurrent execution of the collector and the mutator, an incremental collector uses *barriers*. Barriers are checks in the interpreter that test whether some operation breaks an invariant from the collector. If so, it notifies the collector so that it can restore that invariant. Those checks add overhead to the interpreter. Moreover, the control for an incremental collector is more complex than for a stop-the-world collector.

Generational collection. To improve performance, Lua 5.2 introduced a *generational collector*. The motivation for a generational collector comes from the observation that, typically, most objects die young [25]. Based on this observation, a generational collector concentrates its work on young objects. Roughly, the objects are sorted in two (or more) generations. As they survive garbage-collection cycles, they move to older generations. In a *minor collection* — the norm — the collector assumes that all old objects are alive, and therefore need not be visited; the collection takes a fraction of the time of a regular collection. Moreover, as minor collections are fast, they need not be incremental. If the collector detects too many old objects, then it runs a *major collection*, where it traverses all objects, young and old. Ideally, major collections should be rare or even unnecessary.

The collector in Lua 5.2 actually had two modes: It had this new generational mode, but it also kept the previous incremental mode. Several parts of the implementation are shared by the two modes. In particular, they share the same barriers. In the incremental collector, the barriers detect the creation of links from *black* objects — objects that were already traversed by the collector — to *white* objects — objects not yet touched by the collector. In the generational mode, the same barriers detect the creation of links from old objects to young objects.

While the main invariant for the incremental mode is that black objects do not link to white objects, the main invariant for the generational mode is that old objects do not link to young objects. (As old objects only link to old objects and old objects are not collected in a minor collection, the collector does not need to traverse old objects.) Accordingly, the generational mode simply ensures that all old objects are marked as black, while young object are marked as white. With that invariant, the same barrier works both for the incremental and the generational modes.

Lua uses two kinds of barriers: *back* and *forward*. In a back barrier, the old object becomes *touched*, meaning that it will have to be traversed. In a forward barrier, the young object becomes old. Each particular kind of link has its own policy. For instance, if the link is from an object to its metatable, then the metatable becomes old — a forward barrier — as there is a good chance that the link will not change and that other objects will also link to that same metatable. However, if the link is from a regular table entry, then the table becomes touched — a back barrier — as there is a good chance that other entries in that table will change too. (Once the table is touched, other changes in it will not trigger the barrier.)

Generational collection in practice. The Lua interpreter started in incremental mode, and programs could change the collector mode to generational with a function call. However, the generational mode never became very popular, because it failed to improve the performance of most programs. It took us some time to discover why.

The generational mode in Lua 5.2 used two generations. To keep things simple, any object that survived one collection was considered old. That simple policy created a problem. During the interval between two cycles, inevitably some objects would be created just before the next cycle; therefore, they would still be alive at that cycle and would be promoted to old. So, even if all objects created by a program have short lives, still the program would need major collections from time to time. These frequent major collections hinder the collector.

A possible solution would be to wait two cycles, instead of just one, to move an object to the old generation. Waiting one cycle, the minimum time between the creation of an object and its promotion to old is an epsilon; with two cycles, that minimum time is the time between the two cycles. However, that change largely increases the complexity of the collector, as we will discuss in a moment. So, we removed the generational mode in the next version, Lua 5.3.

Generational collection revisited. In Lua 5.4, we tackled the generational collector again. In this new version, an object waits two cycles before becoming old. When objects live only one cycle as young, the main invariant is trivially preserved: When an object becomes old, all objects it points to also become old. When objects wait two cycles before becoming old, that property is no longer true; an object can become old while pointing to another object that still needs one more cycle to become old. So, the collector must actively control these links.

In the new collector, all objects are created *new*; see Figure 3. After surviving one cycle, they become *survival*. We call them both *young* objects. Young objects can point to any other object, as they will be traversed at the end of the cycle. If a survival object survives another cycle, then it becomes *old-1*. Old-1 objects can still point to survival objects (but not to new objects), so they still must be traversed. After another cycle (that, being old, old-1 objects will survive no matter what), finally the old-1 object becomes really *old*, and then it is not traversed any more.

As we already discussed, the generational mode uses the same barriers used by the incremental mode. If a young object is caught in a forward barrier, it cannot become old immediately, because it can still point to other young objects. Instead, it becomes *old-0*, which in the next cycle becomes old-1. In short, old-0 objects are old — in the sense that they will not be collected in minor collections — but can point to new and survival objects; old-1 are old but cannot point to new objects; and old cannot point to any young object. Moreover, if any old object is caught in a back barrier, it becomes *touched-1* and goes into a special list, to be visited at the end of the cycle. There it evolves to *touched-2*, which can point to survivals but not to new objects. In yet another cycle then it becomes old again.

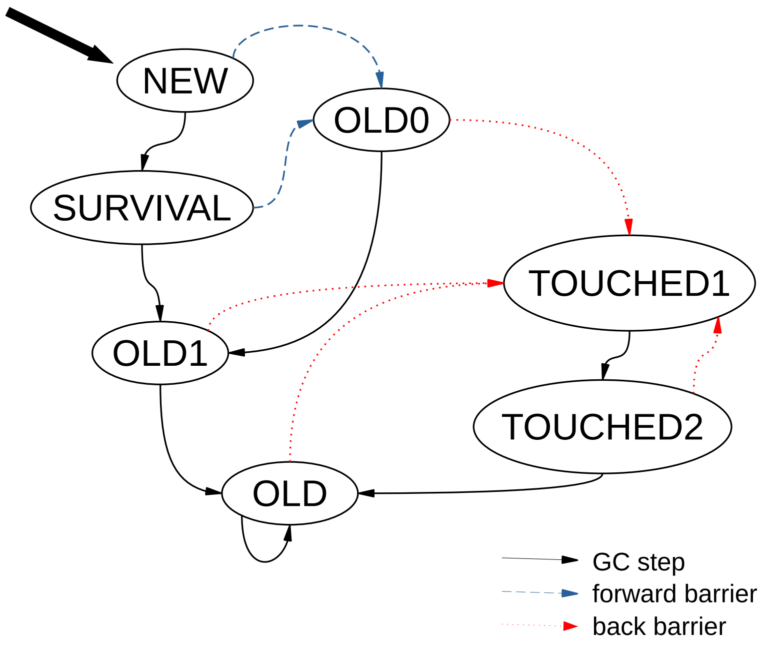

Figure 3: Object states and transitions in generational mode.

So, while the generational collector in Lua 5.2 had three states (young, old, and touched), the collector in Lua 5.4 has seven: new, survival, old-0, old-1, old, touched-1, and touched-2. Clearly, that increases the complexity of the implementation, but not as much as it sounds. The original young state became new and survival, and the original touched state became touched-1 and touched-2. At each cycle, all surviving new objects evolve to survival and all touched-1 objects evolve to touched-2. As they evolve in tandem, their invariants are automatically preserved. The main source of extra complexity was the fact that not all objects become old at once, leading to this scheme with three old states. Overall, the extra complexity is much more in the reasoning — making sure that all cases are covered — than in the code itself.

In return to this extra complexity, we have got a better collector. Many programs do not depend on the performance of the garbage collector, as the time it takes is irrelevant to the total run time. Accordingly, several benchmarks did not benefit from the new collector. For other programs, the garbage collector can have a relevant impact, so these programs can benefit from a better collector. For instance, a simple benchmark with Conway's Game of Life runs 35% faster with the generational mode when compared with the incremental one, using slightly less memory. Another benchmark, which builds and traverses binary trees with different sizes, runs 66% faster using 25% less memory. None of our benchmarks ran slower in generational mode.

One drawback of the generational collector in Lua 5.4 is that major collections are done atomically ("stop-the-world"). The (as of this writing) upcoming Lua 5.5 will fix that, allowing major collections to be done incrementally.

4.5. Finalizers and related features

Since version 2.1, released in 1995, Lua features some mechanism for *object finalization*. The essence of that mechanism has remained unchanged ever since: It allows a *finalizer* function to be associated to an object; when the object is about to be collected, the garbage collector calls that function, with the object as its argument.

Object finalization raises issues of how the program interacts with the collector. At first, only tables could have finalizers, as only tables behaved like "objects", with a clear concept of self. In Lua 3.0, userdata acquired object behavior, and so finalizers could be attached to them, too. Once userdata got finalizers, the two concepts became closely associated: More often than not, finalizers are used to release some external resource, and external resources are represented by userdata inside Lua.

The proper semantics of finalizers has several subtleties, in particular regarding *resurrection*: If a finalizer saves the object being finalized in a global variable or any other external structure, the object is *resurrected* and cannot be collected. Despite the widespread use of this term, it is somewhat misleading: The moment the finalizer is called, the object must already be resurrected; a dead object could not be an argument to the finalizer. Moreover, resurrection is transitive: If object *B* is only referred to by object *A*, and *A* is resurrected, then *B* must also be resurrected. Once the finalizer receives *A*, it can also access *B*.

A consequence of this semantics is that the collector cannot free any object before traversing the complete *transitive closure* of the objects being finalized after a collection. In statically typed languages, the compiler knows beforehand whether an object has a finalizer, due to its type (or class). In a dynamic language like Lua, any object can have a finalizer. That poses a problem for the garbage collector.

Before Lua 4, the collector traversed all objects, separating all dead objects with finalizers. That task was slow. Based on the observation that finalizers are mostly used with userdata, Lua 4.0 restricted finalizers to userdata; so, the collector could traverse only the list of userdata to look for objects to be resurrected.

Both Lua 5.0 and 5.1 kept the restriction that finalizers only worked for userdata. Lua 5.2 removed that restriction. To avoid traversing all objects when looking for resurrection, Lua 5.2 introduced the concept of *marking* objects for finalization. An object is marked for finalization when you set its metatable and the metatable has a finalizer — that is, a `__gc` entry. When this happens, the object is moved to a special internal list. Therefore, when looking for resurrection, the collector has to traverse only this list of marked objects, instead of all objects.

The concept of marking for finalization created an incompatibility: If an object gets a metatable with no finalizer and later the metatable gets a finalizer, Lua 5.1 would call that finalizer when collecting the object, but Lua 5.2 would not. Nevertheless, nobody seemed to care. (At least, we got no reports about it.) In fact, it is not common practice to change the metamethods inside a metatable after the metatable is being actively used.

Ephemeron tables. Besides finalizers, the other main way a program interacts with the garbage collector is through weak references, implemented through *weak tables* in Lua. As we already discussed, Lua offers three kinds of weak tables: tables with weak keys, where the keys are weak; tables with weak values, where the values are weak; and tables that are fully weak, where both keys and values are weak. In all cases, if a key or a value is collected, the whole entry is removed from the table.

Tables with weak keys can create a strange form of cycles that resembles a memory leak. Consider an entry in a table with weak keys with object A as the key and object B as its associated value. If neither A nor B have external links pointing to them, the only way to access them is by traversing the table. Now suppose that B has a reference to A. Because the values in the table are strong, B will be visited by the garbage collector and, therefore, A will also be visited. So, the entry (A,B) will not be removed from the table, even with no other object referring to A or B.

To make the discussion more concrete, let us assume for a moment that there is no way to traverse a table: The only operations on tables are get and set. With that assumption, let us now consider a global variable T containing a table with weak keys and with an entry (A,A). There are no other references to A anywhere else in the program. Following the semantics of weak keys, there is a strong reference from T to A, because A is a value of T; therefore, A cannot be collected and that entry cannot be removed from T. However, the only way to access that entry would be through the key A, which is not present anywhere else. In the end, it is impossible for the program to access that entry — even to remove it — so technically A is garbage, but the collector cannot collect it.

A similar situation happens when the key and the value are different objects, say (A,B), the value B has a reference to the key A, and there are no other references to A or B. Again, the entry would be garbage, given that the program cannot access it, but the collector would collect neither A nor B.

Note that the previous discussion applies regardless of the keys being weak. If a language offers tables without a mechanism to traverse them, it needs a garbage collector that does not handle table entries just as independent references to the key and the value. Otherwise, any entry with a lost key could cause a memory leak.

As tables in Lua do have a traversal mechanism, it is possible for the program to access any entry in a table. So, formally, the entries in those previous examples would not be garbage. However, the whole idea of a key being weak is that it can be removed even if it is not garbage in a strict sense.

To solve this problem, Lua 5.2 introduced *ephemeron tables* [26], an adaptation for tables of the concept of ephemerons [14]. An ephemeron table has weak keys. For its values, it considers an entry (A, B) as equivalent to a reference from A to B, instead of considering B strong. So, if there are no external references to the key A nor to the value B, the entry will be removed. The case where B points to A becomes similar to a cycle, with B pointing to A and A "pointing" (through the table) to B. Like any cycle, its objects will be collected if there are no external references to them.

At first, we considered adding ephemeron tables as a new kind of weak table in Lua. Later, however, we started seeing its semantics as the right semantics for tables with weak keys. As discussed, we can always access a value in a table without knowing its corresponding key, by traversing the table. Nevertheless, we could not find any use case that would benefit from the original semantics, where a weak key is not collected when only its strong value refers to it. So, Lua 5.2 incorporated the ephemeron mechanism as a new semantics for tables with weak keys, instead of a new kind of table.

Deterministic finalization. An important limitation of finalizers is their nondeterminism, that is, the uncertainty about when a finalizer will be called. Often programmers want *deterministic finalization*, that is, the certainty at some point in the program that a given finalizer has been called. An example of a mechanism that can provide deterministic finalization is a `finally` block in Java and Python. Lua 5.4 introduced a somewhat similar mechanism in the form of *to-be-closed* variables.

A key aspect in the design of Lua is its integration with C [9]. Most facilities of the language are available to its API with C. A syntactic construct like `try...finally` is not particularly amenable to be exposed through an API. For that reason, Lua based its mechanism of deterministic finalization upon values. Syntactically, all the mechanism called for was the introduction of *attributes* to the declaration of local variables:

```
local file <close> = ...
```

The annotation `<close>` declares `file` as a *to-be-closed* variable. We took the opportunity to also introduce constants to the language, with just another attribute:

```
local two <const> = 2
two = 4   -- error: attempt to assign to const variable 'two'
```

(Neither `close` nor `const` are reserved words, since they appear only as attributes.)

Once a variable is declared as to-be-closed, its value is *closed* whenever the variable goes out of scope: That includes normal block termination, exiting its block by `break`, `goto`, or `return`, or exiting by an error. To *close* a variable means to call its `__close` metamethod.

To make the mechanism more robust, the `<close>` annotation adds two safeguards: First, the program raises an error if the value being assigned to the variable does not have a `__close` metamethod, unless that value is a false value (`nil` or `false`). Second, it makes the variable constant, that is, the program cannot assign another value to the variable.

Unlike a real finally block, a to-be-closed variable is always associated to a specific object. In practice, more often than not a finally block is attached to some specific resource that is represented by some specific object. Nonetheless, nothing stops a programmer from creating a dummy variable and attaching their particular finally block to its `__close` metamethod. Lexical scoping ensures the metamethod has access to all values that would be available to the block.

A major advantage of this design for Lua is its interface with C. Unlike a syntactic construct like a `finally` block, to-be-closed variables integrate smoothly with the C API. The `<close>` annotation becomes a function that attaches that attribute to a stack slot. (A stack slot is the correspondent to a local variable in the Lua–C API.) Like the `<close>` annotation, the call raises an error if the value at that slot does not have a `__close` metamethod. As soon as that stack slot is popped — when the C function returns, or there is an explicit pop operation, or there is an error — the interpreter calls the `__close` metamethod.

Coroutines present a particularly interesting use of to-be-closed variables. Suppose a coroutine acquires some resources, then it yields, and it is never resumed again. (For instance, the program may decide to kill it.) In that scenario, those resources would be locked indefinitely. Lua 5.4 provides the function `coroutine.close`: It closes all pending to-be-closed variables of the coroutine, thus releasing their resources, and then puts the coroutine in a dead state.

5. Other landmarks

Besides technical quality, social issues are also very relevant in the life of a programming language that is widely used. In particular, an extensive online presence, the production of quality technical literature, and the organization of user events for exchanging ideas and experiences are all quite important for supporting a community of programmers. Here we comment on some landmarks in the history of Lua after the HOPL paper [3].

LuaJIT. In September 2005, Mike Pall announced the initial public release of LuaJIT, a just-in-time compiler for Lua 5.1 targeting x86 CPUs. LuaJIT has evolved into a solid product and now supports several major CPUs and all major compilers and operating systems.

LuaJIT is a technical feat; it combines a high-speed interpreter, hand-crafted in assembly, with a state-of-the-art JIT compiler to deliver impressive performance. LuaJIT proved to be a success in several popular products, which helped increase the wide use of Lua. Unfortunately, LuaJIT chose to stay mostly on Lua 5.1. In particular, LuaJIT refused to implement the new lexical scheme for globals described in §3.2, thus breaking the compatibility with future versions of Lua. That unfortunate decision has been a source of noise for Lua users.

Many users seem to go to LuaJIT lured by the promise of high performance at no programming cost to them. However, LuaJIT was never a drop-in replacement for Lua because LuaJIT relies on a quite different performance model: While Lua relies on C libraries for expensive tasks, LuaJIT relies on specific programming patterns in pure Lua programs. Because LuaJIT uses a trace compiler, C functions do not improve its performance: Quite the opposite, they worsen its performance, because they break traces.

Because of its incompatible performance model, since the beginning LuaJIT was more a fork of Lua than a drop-in replacement. Moreover, LuaJIT added several extensions to the language: Besides access to external functions, its FFI library allows the manipulation of C-like data structures inside Lua code. A program using any of these features cannot run in standard Lua.

Products that need high performance and can afford to code their Lua programs using LuaJIT patterns can benefit greatly. For ordinary products and users, the performance of the standard implementation of Lua seems more than adequate, as witnessed by its success in the game industry.

The impact of perceived performance models on programming is well illustrated by this anecdote:

> Dennis Ritchie encouraged modularity by telling all and sundry that function calls were really, really cheap in C. Everybody started writing small functions and modularizing. Years later we found out that function calls were still expensive on the PDP-11, and VAX code was often spending 50% of its time in the CALLS instruction. Dennis had lied to us! But it was too late; we were all hooked...
>
> — Steve Johnson (quoted by Raymond [27])

Programming in Lua. To reach the masses, a programming language needs a good book. The book *Programming in Lua* first appeared in 2003 and was aimed at Lua 5.0. Although somewhat outdated now, the first edition is still relevant for Lua programmers and is freely available. Later editions appeared as new versions of Lua were released. Along the years, different editions have been translated to German, Japanese, Chinese, Korean, Russian, and Portuguese. The fourth and current edition appeared in 2016 and covers Lua 5.3. The book turned to be a good success, with more than 50,000 copies sold.

Lua Programming Gems. In November 2006 we posted a call for contributions to a book of articles recording some of the wisdom and practice on how to program well in Lua. We had many submissions, which were reviewed by the Lua team. *Lua Programming Gems* was published in December 2008 with 28 articles. The book was self published, like *Programming in Lua*. In October 2022, we made it freely available. The book was a mild success, with more than 4,000 copies sold.

User forums. Spaces where users can meet (virtually or in person) and discuss Lua are very important. One of the focal points of the Lua community is our mailing list, created in February 1997. The discussions held there have been a constant source of motivation and suggestions for improving Lua. There is also a mailing list for discussions in Portuguese created in August 2009. Although mailing lists belong mostly to the previous century, they are still quite useful. Modern meeting places include Stack Overflow (both in English and in Portuguese) prominently.

Lua Workshops. Since 2005, we have organized international workshops on Lua to encourage the Lua community to get together and meet in person and talk about the language, its uses, and its implementation. These events are small, cozy, and quite enjoyable — and always technically strong. They serve the dual purpose of exhibiting Lua projects and discussing the future of Lua. Several changes reported here were first presented at a Lua workshop and have benefited from the feedback we got there. The workshops are also a nice opportunity to meet enthusiastic members of the Lua community.

The first Lua workshop was held in 2005 at Adobe's headquarters in California. It was then that we learned about how Lua got into the game industry [3, page 2-9]. After that, we have had workshops almost every year. We learned about LuaTEX in the 2006 workshop, hosted by Océ in The Netherlands. In several workshops, we saw a demonstration of the Crazy Ivan robot, which is controlled by Lua and has won several competitions. In 2009, we held the workshop at PUC-Rio in Brazil. There we learned about the use of Lua in the game *World of Warcraft* and we could boast having 10 million Lua users! The 2009 workshop also coincided with the launch of Lua BR, a mailing list for discussing Lua in Portuguese. The workshop was held again at PUC-Rio in 2023 to commemorate 30 years of Lua.

The introduction of integers — the biggest change in Lua since version 5.0 — was discussed in several workshops. In the 2011 workshop, held in Switzerland, we already discussed the problem of how to represent 64-bit values in Lua. In 2012, at Verisign's headquarters in Virginia, the discussion was explicitly about how to introduce integers in Lua. We presented the different alternatives we had considered and discussed the main design decisions that later guided the implementation. Finally, in 2014, at the Mail.Ru headquarters in Moscow, we did a full presentation of how integers were introduced, a few months before the release of Lua 5.3.

The Lua website hosts information about all those past workshops, as well as abstracts and slides for most talks presented there [28].

Recognition. The spread of a language is also reflected in various forms of recognition. In 2009 the book *Masterminds of Programming: Conversations with the Creators of Major Programming Languages* [29] included an interview with the Lua team. In January 2012 Lua won the Front Line Award 2011 from Game Developers Magazine in the category Programming Tools. Here is the citation:

> Lua has become an extremely popular programming language, so much so that it's achieved a critical mass of developers in the game industry, meaning Lua skills are transferable from company to company. That's partly due to its speed and the ease with which developers can embed Lua into a game engine. Lua is also highly extensible – it's simple to expand its functionality with libraries either written in Lua, or as extensions in other languages. And it's relatively small and simple, both in terms of the source files, and the resultant code and run-time memory usage.

In 2018, the TEX Users Group (TUG) held its annual conference in Rio de Janeiro [30] with the Lua team as special guests in recognition of the role of Lua in modern TEX engines. LuaTeX is an extended version of pdfTeX that uses Lua as an embedded scripting language: It features access to the internals of TeX from Lua. Moreover, large parts of the TeX engine were replaced by Lua code in LuaTeX. The TeX Live distribution adopted LuaTeX as a successor to pdfTeX and so LuaTeX is widely distributed along TeX.

Local recognition is also important. In 2017, Lua featured in the "Inovanças – Creations Brazilian style" exhibition at the Museum of Tomorrow in Rio. In 2022 the Lua team received the Pedro Ernesto Medal, the highest decoration given by the city of Rio de Janeiro.

6. Conclusion

In these 30+ years, Lua has evolved to meet requirements coming from outside, ones that we had not planned for or even imagined. Our focus on keeping the language small and its implementation portable has helped us meeting these requirements. Lua has acquired several modern features without losing its original character. We hope to continue in this path.

Most changes in the last 15 years can be seen as incremental, several of them improving non-conventional features introduced in previous versions. This is the case for the generational collector (Lua 5.2 and 5.4), which is an improvement on the incremental collector (5.1); ephemeron tables (5.2), which are an improvement on weak tables (5.0); and yieldable C calls (5.2), which are an improvement on coroutines (5.0). The introduction of integers, too, can be seen as mostly an incremental change: Except for the bitwise operators, integers brought remarkably little change in how programmers use Lua. A few changes can be seen as design fixes, often for features recently introduced. The lexical scheme for global variables (5.2) can be seen as a fix for the somewhat unhampered mechanism of function environments from Lua 5.0. Finally, a few changes brought real novelties to the language, such as to-be-closed variables. Nonetheless, the changes accumulated in the Lua 5 series have given Lua a stronger personality, making it a more interesting language.

Acknowledgments

We are very grateful to all Lua users and all companies that have adopted Lua for their products: They are a continuous motivation for improving Lua. Special thanks go to the members of our mailing list, for their ideas and suggestions, which are generously shared in a friendly environment. We thank the guest editors Francisco Sant'Anna and Sérgio Medeiros for organizing this special issue to celebrate 30 years of Lua. We thank the referees for their insightful comments and suggestions. We also thank PUC-Rio and IMPA for their continued support of our work on Lua. Special thanks go to the members of Tecgraf, where Lua was born, and LabLua, where Lua is developed. This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brazil (CAPES)—Finance Code 001.

References

[1] Wikipedia contributors, List of applications using Lua — Wikipedia, the free encyclopedia, https://en.wikipedia.org/w/index.php?title=List_of_applications_using_Lua&oldid=1226927748, [Online; accessed 17-September-2024] (Sep. 2024).

[2] Wikipedia contributors, Category:Lua-scripted video games, https://en.wikipedia.org/w/index.php?title=Category:Lua_(programming_language)-scripted_video_games&oldid=1119515244, [Online; accessed 27-September-2024] (Sep. 2024).

[3] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, The evolution of Lua, in: HOPL III, Third ACM SIGPLAN Conference on History of Programming Languages, ACM, 2007, pp. 2–1–2–26. doi:10.1145/1238844.1238846.

[4] N. Wirth, A plea for lean software, Computer 28 (2) (1995) 64–68. doi:10.1109/2.348001.

[5] B. Hubert, Why bloat is still software's biggest vulnerability: A 2024 plea for lean software, IEEE Spectrum 61 (4) (2024) 22–50. doi:10.1109/MSPEC.2024.10491389.

[6] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, A look at the design of Lua, Communications of the ACM 61 (11) (2018) 114–123. doi:10.1145/3186277.

[7] A. Hirschi, Traveling Light, the Lua Way , IEEE Software 24 (05) (2007) 31–38. doi:10.1109/MS.2007.150.
URL https://doi.ieeecomputersociety.org/10.1109/MS.2007.150

[8] R. Ierusalimschy, Programming with multiple paradigms in Lua, in: S. Escobar (Ed.), 18th Int'l Workshop on Functional and (Constraint) Logic Programming, Springer, Heidelberg, Germany, 2009, pp. 5–13, (LNCS, volume 5979). doi:10.1007/978-3-642-11999-6_1.

[9] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, Passing a language through the eye of a needle, Communications of the ACM 54 (7) (2011) 38–43. doi:10.1145/1965724.1965739.

[10] A. L. de Moura, N. Rodriguez, R. Ierusalimschy, Coroutines in Lua, Journal of Universal Computer Science 10 (7) (2004) 910–925. doi:10.3217/JUCS-010-07-0910.

[11] A. L. de Moura, R. Ierusalimschy, Revisiting coroutines, ACM Transactions on Programming Languages and Systems 31 (2) (2009) 6.1–6.31. doi:10.1145/1462166.1462167.

[12] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, Lua 5.4 Reference Manual, Lua.org, 2020.
URL https://www.lua.org/manual/5.4/

[13] R. Ierusalimschy, Programming in Lua, 4th Edition, Lua.org, 2016.

[14] B. Hayes, Ephemerons: a new finalization mechanism, in: Proc. of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, 1997, pp. 176–183.

[15] E. W. Dijkstra, Letters to the editor: go to statement considered harmful, Communications of the ACM 11 (3) (1968) 147–148. doi:10.1145/362929.362947.

[16] D. E. Knuth, Structured programming with go to statements, ACM Computing Surveys 6 (4) (1974) 261–301. doi:10.1145/356635.356640.

[17] M. Pall, ResumableVmPatch, http://lua-users.org/wiki/ResumableVmPatch, [Online; accessed 10-October-2024] (2005).

[18] R. Ierusalimschy, A text pattern-matching tool based on Parsing Expression Grammars, Software: Practice and Experience 39 (3) (2009) 221–258. doi:10.1002/spe.892.

[19] IEEE, IEEE Standard for Floating-Point Arithmetic, std 754-2019 (Jul. 2019). doi:10.1109/IEEESTD.2019.8766229.

[20] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, The implementation of Lua 5.0, Journal of Universal Computer Science 11 (7) (2005) 1159–1176. doi:doi.org/10.3217/jucs-011-07-1159.

[21] R. Ierusalimschy, What is next for Lua? a personal perspective, Presented at the Lua Workshop 2012 (2012).
URL https://www.lua.org/wshop12.html

[22] ISO, International Standard: Programming languages — C, ISO/IEC 9899:1999(E) (2000).

[23] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, Lua 5.1 Reference Manual, Lua.org, 2006.
URL https://www.lua.org/manual/5.1/

[24] L. Vieira Neto, R. Ierusalimschy, A. L. de Moura, M. Balmer, Scriptable operating systems with Lua, ACM SIGPLAN Notices 50 (2) (2014) 2–10. doi:10.1145/2775052.2661096.

[25] R. Jones, A. Hosking, E. Moss, The Garbage Collection Handbook: The Art of Automatic Memory Management, Chapman and Hall/CRC, 2023. doi:10.1201/9781003276142.

[26] A. Barros, R. Ierusalimschy, Eliminating cycles in weak tables, Journal of Universal Computer Science 14 (21) (2008) 3481–3497. doi:10.3217/jucs-014-21-3481.

[27] E. S. Raymond, The Art of UNIX Programming, Pearson Education, 2003.

[28] Lua, Community, https://www.lua.org/community.html, [Online; accessed 19-October-2024] (2024).

[29] F. Biancuzzi, S. Warden, Masterminds of Programming: Conversations with the Creators of Major Programming Languages, O'Reilly, 2009.

[30] TEX Users Group, TUG 2018 Conference Proceedings, TUGboat 39 (2) (Jul. 2018).