The Evolution of Lua

Roberto Ierusalimschy

Department of Computer Science,
PUC-Rio, Rio de Janeiro, Brazil

roberto@inf.puc-rio.br

Luiz Henrique de Figueiredo

IMPA–Instituto Nacional de
Matemática Pura e Aplicada, Brazil

lhf@impa.br

Waldemar Celes

Department of Computer Science,
PUC-Rio, Rio de Janeiro, Brazil

celes@inf.puc-rio.br

Abstract

We report on the birth and evolution of Lua and discuss how it moved from a simple configuration language to a versatile, widely used language that supports extensible semantics, anonymous functions, full lexical scoping, proper tail calls, and coroutines.

Categories and Subject Descriptors K.2 [*HISTORY OF COMPUTING*]: Software; D.3 [*PROGRAMMING LANGUAGES*]

1. Introduction

Lua is a scripting language born in 1993 at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Since then, Lua has evolved to become widely used in all kinds of industrial applications, such as robotics, literate programming, distributed business, image processing, extensible text editors, Ethernet switches, bioinformatics, finite-element packages, web development, and more [2]. In particular, Lua is one of the leading scripting languages in game development.

Lua has gone far beyond our most optimistic expectations. Indeed, while almost all programming languages come from North America and Western Europe (with the notable exception of Ruby, from Japan) [4], Lua is the only language created in a developing country to have achieved global relevance.

From the start, Lua was designed to be simple, small, portable, fast, and easily embedded into applications. These design principles are still in force, and we believe that they account for Lua's success in industry. The main characteristic of Lua, and a vivid expression of its simplicity, is that it offers a single kind of data structure, the *table*, which is the Lua term for an associative array [9]. Although most scripting languages offer associative arrays, in no other language do associative arrays play such a central role. Lua tables provide simple and efficient implementations for modules, prototype-based objects, class-based objects, records, arrays, sets, bags, lists, and many other data structures [28].

In this paper, we report on the birth and evolution of Lua. We discuss how Lua moved from a simple configuration language to a powerful (but still simple) language that supports extensible semantics, anonymous functions, full lexical scoping, proper tail calls, and coroutines. In §2 we give an overview of the main concepts in Lua, which we use in the other sections to discuss how Lua has evolved. In §3 we relate the prehistory of Lua, that is, the setting that led to its creation. In §4 we relate how Lua was born, what its original design goals were, and what features its first version had. A discussion of how and why Lua has evolved is given in §5. A detailed discussion of the evolution of selected features is given in §6. The paper ends in §7 with a retrospective of the evolution of Lua and in §8 with a brief discussion of the reasons for Lua's success, especially in games.

2. Overview

In this section we give a brief overview of the Lua language and introduce the concepts discussed in §5 and §6. For a complete definition of Lua, see its reference manual [32]. For a detailed introduction to Lua, see Roberto's book [28]. For concreteness, we shall describe Lua 5.1, which is the current version at the time of this writing (April 2007), but most of this section applies unchanged to previous versions.

Syntactically, Lua is reminiscent of Modula and uses familiar keywords. To give a taste of Lua's syntax, the code below shows two implementations of the factorial function, one recursive and another iterative. Anyone with a basic knowledge of programming can probably understand these examples without explanation.

```
function factorial(n)          function factorial(n)
  if n == 0 then                 local a = 1
    return 1                     for i = 1,n do
  else                             a = a*i
    return n*factorial(n-1)       end
  end                            return a
end                            end
```

Semantically, Lua has many similarities with Scheme, even though these similarities are not immediately clear because the two languages are syntactically very different. The influence of Scheme on Lua has gradually increased during Lua's evolution: initially, Scheme was just a language in the background, but later it became increasingly important as a source of inspiration, especially with the introduction of anonymous functions and full lexical scoping.

Like Scheme, Lua is dynamically typed: variables do not have types; only values have types. As in Scheme, a variable in Lua never contains a structured value, only a reference to one. As in Scheme, a function name has no special status in Lua: it is just a regular variable that happens to refer to a function value. Actually, the syntax for function definition 'function foo() ⋯ end' used above is just syntactic sugar for the assignment of an anonymous function to a variable: 'foo = function () ⋯ end'. Like Scheme, Lua has first-class functions with lexical scoping. Actually, all values in Lua are first-class values: they can be assigned to global and local variables, stored in tables, passed as arguments to functions, and returned from functions.

One important semantic difference between Lua and Scheme—and probably the main distinguishing feature of Lua—is that Lua offers *tables* as its sole data-structuring mechanism. Lua tables are associative arrays [9], but with some important features. Like all values in Lua, tables are first-class values: they are not bound to specific variable names, as they are in Awk and Perl. A table can have any value as key and can store any value. Tables allow simple and efficient implementation of records (by using field names as keys), sets (by using set elements as keys), generic linked structures, and many other data structures. Moreover, we can use a table to implement an array by using natural numbers as indices. A careful implementation [31] ensures that such a table uses the same amount of memory that an array would (because it is represented internally as an actual array) and performs better than arrays in similar languages, as independent benchmarks show [1].

Lua offers an expressive syntax for creating tables in the form of *constructors*. The simplest constructor is the expression '{}', which creates a new, empty table. There are also constructors to create lists (or arrays), such as

```
{"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}
```

and to create records, such as

```
{lat= -22.90, long= -43.23, city= "Rio de Janeiro"}
```

These two forms can be freely mixed. Tables are indexed using square brackets, as in 't[2]', with 't.x' as sugar for 't["x"]'.

The combination of table constructors and functions turns Lua into a powerful general-purpose procedural data-description language. For instance, a bibliographic database in a format similar to the one used in BibTEX [34] can be written as a series of table constructors such as this:

```
article{"spe96",
  authors = {"Roberto Ierusalimschy",
             "Luiz Henrique de Figueiredo",
             "Waldemar Celes"},
  title = "Lua: an Extensible Extension Language",
  journal = "Software: Practice & Experience",
  year = 1996,
}
```

Although such a database seems to be an inert data file, it is actually a valid Lua program: when the database is loaded into Lua, each item in it invokes a function, because 'article{⋯}' is syntactic sugar for 'article({⋯})', that is, a function call with a table as its single argument. It is in this sense that such files are called procedural data files.

We say that Lua is an extensible extension language [30]. It is an *extension language* because it helps to extend applications through configuration, macros, and other end-user customizations. Lua is designed to be embedded into a host application so that users can control how the application behaves by writing Lua programs that access application services and manipulate application data. It is *extensible* because it offers *userdata* values to hold application data and *extensible semantics* mechanisms to manipulate these values in natural ways. Lua is provided as a small core that can be extended with user functions written in both Lua and C. In particular, input and output, string manipulation, mathematical functions, and interfaces to the operating system are all provided as external libraries.

Other distinguishing features of Lua come from its implementation:

Portability: Lua is easy to build because it is implemented in strict ANSI C.[1] It compiles out-of-the-box on most platforms (Linux, Unix, Windows, Mac OS X, etc.), and runs with at most a few small adjustments in virtually all platforms we know of, including mobile devices (e.g., handheld computers and cell phones) and embedded microprocessors (e.g., ARM and Rabbit). To ensure portability, we strive for warning-free compilations under as many compilers as possible.

Ease of embedding: Lua has been designed to be easily embedded into applications. An important part of Lua is a well-defined application programming interface (API) that allows full communication between Lua code and external code. In particular, it is easy to extend Lua by exporting C functions from the host application. The API allows Lua to interface not only with C and C++, but also with other languages, such as Fortran, Java, Smalltalk, Ada, C# (.Net), and even with other scripting languages (e.g., Perl and Ruby).

---

[1] Actually, Lua is implemented in "clean C", that is, the intersection of C and C++. Lua compiles unmodified as a C++ library.

Small size: Adding Lua to an application does not bloat it. The whole Lua distribution, including source code, documentation, and binaries for some platforms, has always fit comfortably on a floppy disk. The tarball for Lua 5.1, which contains source code, documentation, and examples, takes 208K compressed and 835K uncompressed. The source contains around 17,000 lines of C. Under Linux, the Lua interpreter built with all standard Lua libraries takes 143K. The corresponding numbers for most other scripting languages are more than an order of magnitude larger, partially because Lua is primarily meant to be embedded into applications and so its official distribution includes only a few libraries. Other scripting languages are meant to be used standalone and include many libraries.

Efficiency: Independent benchmarks [1] show Lua to be one of the fastest languages in the realm of interpreted scripting languages. This allows application developers to write a substantial fraction of the whole application in Lua. For instance, over 40% of Adobe Lightroom is written in Lua (that represents around 100,000 lines of Lua code).

Although these are features of a specific implementation, they are possible only due to the design of Lua. In particular, Lua's simplicity is a key factor in allowing a small, efficient implementation [31].

3. Prehistory

Lua was born in 1993 inside Tecgraf, the Computer Graphics Technology Group of PUC-Rio in Brazil. The creators of Lua were Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Roberto was an assistant professor at the Department of Computer Science of PUC-Rio. Luiz Henrique was a post-doctoral fellow, first at IMPA and later at Tecgraf. Waldemar was a Ph.D. student in Computer Science at PUC-Rio. All three were members of Tecgraf, working on different projects there before getting together to work on Lua. They had different, but related, backgrounds: Roberto was a computer scientist interested mainly in programming languages; Luiz Henrique was a mathematician interested in software tools and computer graphics; Waldemar was an engineer interested in applications of computer graphics. (In 2001, Waldemar joined Roberto as faculty at PUC-Rio and Luiz Henrique became a researcher at IMPA.)

Tecgraf is a large research and development laboratory with several industrial partners. During the first ten years after its creation in May 1987, Tecgraf focused mainly on building basic software tools to enable it to produce the interactive graphical programs needed by its clients. Accordingly, the first Tecgraf products were drivers for graphical terminals, plotters, and printers; graphical libraries; and graphical interface toolkits. From 1977 until 1992, Brazil had a pol-

icy of strong trade barriers (called a "market reserve") for computer hardware and software motivated by a nationalistic feeling that Brazil could and should produce its own hardware and software. In that atmosphere, Tecgraf's clients could not afford, either politically or financially, to buy customized software from abroad: by the market reserve rules, they would have to go through a complicated bureaucratic process to prove that their needs could not be met by Brazilian companies. Added to the natural geographical isolation of Brazil from other research and development centers, those reasons led Tecgraf to implement from scratch the basic tools it needed.

One of Tecgraf's largest partners was (and still is) Petrobras, the Brazilian oil company. Several Tecgraf products were interactive graphical programs for engineering applications at Petrobras. By 1993, Tecgraf had developed little languages for two of those applications: a data-entry application and a configurable report generator for lithology profiles. These languages, called DEL and SOL, were the ancestors of Lua. We describe them briefly here to show where Lua came from.

3.1 DEL

The engineers at Petrobras needed to prepare input data files for numerical simulators several times a day. This process was boring and error-prone because the simulation programs were legacy code that needed strictly formatted input files — typically bare columns of numbers, with no indication of what each number meant, a format inherited from the days of punched cards. In early 1992, Petrobras asked Tecgraf to create at least a dozen graphical front-ends for this kind of data entry. The numbers would be input interactively, just by clicking on the relevant parts of a diagram describing the simulation — a much easier and more meaningful task for the engineers than editing columns of numbers. The data file, in the correct format for the simulator, would be generated automatically. Besides simplifying the creation of data files, such front-ends provided the opportunity to add data validation and also to compute derived quantities from the input data, thus reducing the amount of data needed from the user and increasing the reliability of the whole process.

To simplify the development of those front-ends, a team led by Luiz Henrique de Figueiredo and Luiz Cristovão Gomes Coelho decided to code all front-ends in a uniform way, and so designed DEL ("data-entry language"), a simple declarative language to describe each data-entry task [17]. DEL was what is now called a domain-specific language [43], but was then simply called a little language [10].

A typical DEL program defined several "entities". Each entity could have several fields, which were named and typed. For implementing data validation, DEL had predicate statements that imposed restrictions on the values of entities. DEL also included statements to specify how data was to be input and output. An entity in DEL was essentially what is called a structure or record in conventional

programming languages. The important difference — and what made DEL suitable for the data-entry problem — is that entity names also appeared in a separate graphics metafile, which contained the associated diagram over which the engineer did the data entry. A single interactive graphical interpreter called ED (an acronym for 'entrada de dados', which means 'data entry' in Portuguese) was written to interpret DEL programs. All those data-entry front-ends requested by Petrobras were implemented as DEL programs that ran under this single graphical application.

DEL was a success both among the developers at Tecgraf and among the users at Petrobras. At Tecgraf, DEL simplified the development of those front-ends, as originally intended. At Petrobras, DEL allowed users to tailor data-entry applications to their needs. Soon users began to demand more power from DEL, such as boolean expressions for controlling whether an entity was active for input or not, and DEL became heavier. When users began to ask for control flow, with conditionals and loops, it was clear that ED needed a real programming language instead of DEL.

3.2 SOL

At about the same time that DEL was created, a team lead by Roberto Ierusalimschy and Waldemar Celes started working on PGM, a configurable report generator for lithology profiles, also for Petrobras. The reports generated by PGM consisted of several columns (called "tracks") and were highly configurable: users could create and position the tracks, and could choose colors, fonts, and labels; each track could have a grid, which also had its set of options (log/linear, vertical and horizontal ticks, etc.); each curve had its own scale, which had to be changed automatically in case of overflow; etc. All this configuration was to be done by the end-users, typically geologists and engineers from Petrobras working in oil plants and off-shore platforms. The configurations had to be stored in files, for reuse. The team decided that the best way to configure PGM was through a specialized description language called SOL, an acronym for *Simple Object Language*.

Because PGM had to deal with many different objects, each with many different attributes, the SOL team decided not to fix those objects and attributes into the language. Instead, SOL allowed type declarations, as in the code below:

```
type @track{ x:number, y:number=23, id=0 }
type @line{ t:@track=@track{x=8}, z:number* }
T = @track{ y=9, x=10, id="1992-34" }
L = @line{ t=@track{x=T.y, y=T.x}, z=[2,3,4] }
```

This code defines two types, `track` and `line`, and creates two objects, a track T and a line L. The `track` type contains two numeric attributes, x and y, and an untyped attribute, `id`; attributes y and `id` have default values. The `line` type contains a track t and a list of numbers z. The track t has as default value a track with x=8, y=23, and id=0. The syntax

of SOL was strongly influenced by BibTEX [34] and UIL, a language for describing user interfaces in Motif [39].

The main task of the SOL interpreter was to read a report description, check whether the given objects and attributes were correctly typed, and then present the information to the main program (PGM). To allow the communication between the main program and the SOL interpreter, the latter was implemented as a C library that was linked to the main program. The main program could access all configuration information through an API in this library. In particular, the main program could register a callback function for each type, which the SOL interpreter would call to create an object of that type.

4. Birth

The SOL team finished an initial implementation of SOL in March 1993, but they never delivered it. PGM would soon require support for procedural programming to allow the creation of more sophisticated layouts, and SOL would have to be extended. At the same time, as mentioned before, ED users had requested more power from DEL. ED also needed further descriptive facilities for programming its user interface. Around mid-1993, Roberto, Luiz Henrique, and Waldemar got together to discuss DEL and SOL, and concluded that the two languages could be replaced by a single, more powerful language, which they decided to design and implement. Thus the Lua team was born; it has not changed since.

Given the requirements of ED and PGM, we decided that we needed a real programming language, with assignments, control structures, subroutines, etc. The language should also offer data-description facilities, such as those offered by SOL. Moreover, because many potential users of the language were not professional programmers, the language should avoid cryptic syntax and semantics. The implementation of the new language should be highly portable, because Tecgraf's clients had a very diverse collection of computer platforms. Finally, since we expected that other Tecgraf products would also need to embed a scripting language, the new language should follow the example of SOL and be provided as a library with a C API.

At that point, we could have adopted an existing scripting language instead of creating a new one. In 1993, the only real contender was Tcl [40], which had been explicitly designed to be embedded into applications. However, Tcl had unfamiliar syntax, did not offer good support for data description, and ran only on Unix platforms. We did not consider LISP or Scheme because of their unfriendly syntax. Python was still in its infancy. In the free, do-it-yourself atmosphere that then reigned in Tecgraf, it was quite natural that we should try to develop our own scripting language. So, we started working on a new language that we hoped would be simpler to use than existing languages. Our original design decisions were: keep the language simple and small, and keep the im-

plementation simple and portable. Because the new language was partially inspired by SOL (*sun* in Portuguese), a friend at Tecgraf (Carlos Henrique Levy) suggested the name 'Lua' (*moon* in Portuguese), and Lua was born. (DEL did not influence Lua as a language. The main influence of DEL on the birth of Lua was rather the realization that large parts of complex applications could be written using embeddable scripting languages.)

We wanted a light full language with data-description facilities. So we took SOL's syntax for record and list construction (but not type declaration), and unified their implementation using tables: records use strings (the field names) as indices; lists use natural numbers. An assignment such as

```
T = @track{ y=9, x=10, id="1992-34" }
```

which was valid in SOL, remained valid in Lua, but with a different meaning: it created an object (that is, a table) with the given fields, and then called the function `track` on this table to validate the object or perhaps to provide default values to some of its fields. The final value of the expression was that table.

Except for its procedural data-description constructs, Lua introduced no new concepts: Lua was created for production use, not as an academic language designed to support research in programming languages. So, we simply borrowed (even unconsciously) things that we had seen or read about in other languages. We did not reread old papers to remember details of existing languages. We just started from what we knew about other languages and reshaped that according to our tastes and needs.

We quickly settled on a small set of control structures, with syntax mostly borrowed from Modula (`while`, `if`, and `repeat until`). From CLU we took multiple assignment and multiple returns from function calls. We regarded multiple returns as a simpler alternative to reference parameters used in Pascal and Modula and to in-out parameters used in Ada; we also wanted to avoid explicit pointers (used in C). From C++ we took the neat idea of allowing a local variable to be declared only where we need it. From SNOBOL and Awk we took associative arrays, which we called tables; however, tables were to be objects in Lua, not attached to variables as in Awk.

One of the few (and rather minor) innovations in Lua was the syntax for string concatenation. The natural '+' operator would be ambiguous, because we wanted automatic coercion of strings to numbers in arithmetic operations. So, we invented the syntax '`..`' (two dots) for string concatenation.

A polemic point was the use of semicolons. We thought that requiring semicolons could be a little confusing for engineers with a Fortran background, but not allowing them could confuse those with a C or Pascal background. In typical committee fashion, we settled on optional semicolons.

Initially, Lua had seven types: numbers (implemented solely as reals), strings, tables, nil, userdata (pointers to C objects), Lua functions, and C functions. To keep the language small, we did not initially include a boolean type: as in Lisp, nil represented *false* and any other value represented *true*. Over 13 years of continuous evolution, the only changes in Lua types were the unification of Lua functions and C functions into a single function type in Lua 3.0 (1997) and the introduction of booleans and threads in Lua 5.0 (2003) (see §6.1). For simplicity, we chose to use dynamic typing instead of static typing. For applications that needed type checking, we provided basic reflective facilities, such as run-time type information and traversal of the global environment, as built-in functions (see §6.11).

By July 1993, Waldemar had finished the first implementation of Lua as a course project supervised by Roberto. The implementation followed a tenet that is now central to Extreme Programming: "the simplest thing that could possibly work" [7]. The lexical scanner was written with `lex` and the parser with `yacc`, the classic Unix tools for implementing languages. The parser translated Lua programs into instructions for a stack-based virtual machine, which were then executed by a simple interpreter. The C API made it easy to add new functions to Lua, and so this first version provided only a tiny library of five built-in functions (`next`, `nextvar`, `print`, `tonumber`, `type`) and three small external libraries (input and output, mathematical functions, and string manipulation).

Despite this simple implementation — or possibly because of it — Lua surpassed our expectations. Both PGM and ED used Lua successfully (PGM is still in use today; ED was replaced by EDG [12], which was mostly written in Lua). Lua was an immediate success in Tecgraf and soon other projects started using it. This initial use of Lua at Tecgraf was reported in a brief talk at the VII Brazilian Symposium on Software Engineering, in October 1993 [29].

The remainder of this paper relates our journey in improving Lua.

5. History

Figure 1 shows a timeline of the releases of Lua. As can be seen, the time interval between versions has been gradually increasing since Lua 3.0. This reflects our perception that

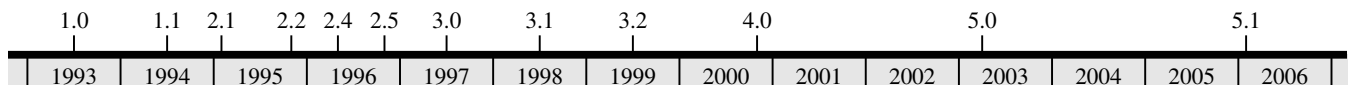| 1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006

Figure 1. The releases of Lua.

	1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
constructors	●	●	●	●	●	●	●	●	●	●	●	●
garbage collection	●	●	●	●	●	●	●	●	●	●	●	●
extensible semantics	○	○	●	●	●	●	●	●	●	●	●	●
support for OOP	○	○	●	●	●	●	●	●	●	●	●	●
long strings	○	○	○	●	●	●	●	●	●	●	●	●
debug API	○	○	○	●	●	●	●	●	●	●	●	●
external compiler	○	○	○	○	●	●	●	●	●	●	●	●
vararg functions	○	○	○	○	○	●	●	●	●	●	●	●
pattern matching	○	○	○	○	○	●	●	●	●	●	●	●
conditional compilation	○	○	○	○	○	○	●	●	●	○	○	○
anonymous functions, closures	○	○	○	○	○	○	○	●	●	●	●	●
debug library	○	○	○	○	○	○	○	○	●	●	●	●
multi-state API	○	○	○	○	○	○	○	○	○	●	●	●
for statement	○	○	○	○	○	○	○	○	○	●	●	●
long comments	○	○	○	○	○	○	○	○	○	○	●	●
full lexical scoping	○	○	○	○	○	○	○	○	○	○	●	●
booleans	○	○	○	○	○	○	○	○	○	○	●	●
coroutines	○	○	○	○	○	○	○	○	○	○	●	●
incremental garbage collection	○	○	○	○	○	○	○	○	○	○	○	●
module system	○	○	○	○	○	○	○	○	○	○	○	●
	1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
libraries	4	4	4	4	4	4	4	4	5	6	8	9
built-in functions	5	7	11	11	13	14	25	27	35	0	0	0
API functions	30	30	30	30	32	32	33	47	41	60	76	79
vm type (stack × register)	S	S	S	S	S	S	S	S	S	S	R	R
vm instructions	64	65	69	67	67	68	69	128	64	49	35	38
keywords	16	16	16	16	16	16	16	16	16	18	21	21
other tokens	21	21	23	23	23	23	24	25	25	25	24	26

Table 1. The evolution of features in Lua.

Lua was becoming a mature product and needed stability for the benefit of its growing community. Nevertheless, the need for stability has not hindered progress. Major new versions of Lua, such as Lua 4.0 and Lua 5.0, have been released since then.

The long times between versions also reflects our release model. Unlike other open-source projects, our alpha versions are quite stable and beta versions are essentially final, except for uncovered bugs.[2] This release model has proved to be good for Lua stability. Several products have been shipped with alpha or beta versions of Lua and worked fine. However, this release model did not give users much chance to experiment with new versions; it also deprived us of timely feedback on proposed changes. So, during the development of Lua 5.0 we started to release "work" versions, which are just snapshots of the current development of Lua. This move brought our current release model closer to the "Release Early, Release Often" motto of the open-source community.

In the remainder of this section we discuss some milestones in the evolution of Lua. Details on the evolution of several specific features are given in §6. Table 1 summarizes this evolution. It also contains statistics about the size of Lua, which we now discuss briefly.

The number of standard libraries has been kept small because we expect that most Lua functions will be provided by the host application or by third-party libraries. Until Lua 3.1, the only standard libraries were for input and output, string manipulation, mathematical functions, and a special library of built-in functions, which did not use the C API but directly accessed the internal data structures. Since then, we have added libraries for debugging (Lua 3.2), interfacing with the operating system (Lua 4.0), tables and coroutines (Lua 5.0), and modules (Lua 5.1).

The size of C API changed significantly when it was redesigned in Lua 4.0. Since then, it has moved slowly toward completeness. As a consequence, there are no longer any built-in functions: all standard libraries are implemented on top the C API, without accessing the internals of Lua.

The virtual machine, which executes Lua programs, was stack-based until Lua 4.0. In Lua 3.1 we added variants

---

[2] The number of bugs found after final versions were released has been consistently small: only 10 in Lua 4.0, 17 in Lua 5.0, and 10 in Lua 5.1 so far, none of them critical bugs.

for many instructions, to try to improve performance. However, this turned out to be too complicated for little performance gain and we removed those variants in Lua 3.2. Since Lua 5.0, the virtual machine is register-based [31]. This change gave the code generator more opportunities for optimization and reduced the number of instructions of typical Lua programs. (Instruction dispatch is a significant fraction of the time spent in the virtual machine [13].) As far as we know, the virtual machine of Lua 5.0 was the first register-based virtual machine to have wide use.

5.1 Lua 1

The initial implementation of Lua was a success in Tecgraf and Lua attracted users from other Tecgraf projects. New users create new demands. Several users wanted to use Lua as the support language for graphics metafiles, which abounded in Tecgraf. Compared with other programmable metafiles, Lua metafiles have the advantage of being based on a truly procedural language: it is natural to model complex objects by combining procedural code fragments with declarative statements. In contrast, for instance, VRML [8] must use another language (Javascript) to model procedural objects.

The use of Lua for this kind of data description, especially large graphics metafiles, posed challenges that were unusual for typical scripting languages. For instance, it was not uncommon for a diagram used in the data-entry program ED to have several thousand parts described by a single Lua table constructor with several thousand items. That meant that Lua had to cope with huge programs and huge expressions. Because Lua precompiled all programs to bytecode for a virtual machine on the fly, it also meant that the Lua compiler had to run fast, even for large programs.

By replacing the `lex`-generated scanner used in the first version by a hand-written one, we almost doubled the speed of the Lua compiler on typical metafiles. We also modified Lua's virtual machine to handle a long constructor by adding key-value pairs to the table in batches, not individually as in the original virtual machine. These changes solved the initial demands for better performance. Since then, we have always tried to reduce the time spent on precompilation.

In July 1994, we released a new version of Lua with those optimizations. This release coincided with the publication of the first paper describing Lua, its design, and its implementation [15]. We named the new version 'Lua 1.1'. The previous version, which was never publicly released, was then named 'Lua 1.0'. (A snapshot of Lua 1.0 taken in July 1993 was released in October 2003 to celebrate 10 years of Lua.)

Lua 1.1 was publicly released as software available in source code by ftp, before the open-source movement got its current momentum. Lua 1.1 had a restrictive user license: it was freely available for academic purposes but commercial uses had to be negotiated. That part of the license did not work: although we had a few initial contacts, no commercial uses were ever negotiated. This and the fact that other

scripting languages (e.g, Tcl) were free made us realize that restrictions on commercial uses might even discourage academic uses, since some academic projects plan to go to market eventually. So, when the time came to release the next version (Lua 2.1), we chose to release it as unrestricted free software. Naively, we wrote our own license text as a slight collage and rewording of existing licenses. We thought it was clear that the new license was quite liberal. Later, however, with the spread of open-source licenses, our license text became a source of noise among some users; in particular, it was not clear whether our license was compatible with GPL. In May 2002, after a long discussion in the mailing list, we decided to release future versions of Lua (starting with Lua 5.0) under the well-known and very liberal MIT license [3]. In July 2002, the Free Software Foundation confirmed that our previous license was compatible with GPL, but we were already committed to adopting the MIT license. Questions about our license have all but vanished since then.

5.2 Lua 2

Despite all the hype surrounding object-oriented programming (which in the early 1990s had reached its peak) and the consequent user pressure to add object-oriented features to Lua, we did not want to turn Lua into an object-oriented language because we did not want to fix a programming paradigm for Lua. In particular, we did not think that Lua needed objects and classes as primitive language concepts, especially because they could be implemented with tables if needed (a table can hold both object data and methods, since functions are first-class values). Despite recurring user pressure, we have not changed our minds to this day: Lua does not force any object or class model onto the programmer. Several object models have been proposed and implemented by users; it is a frequent topic of discussion in our mailing list. We think this is healthy.

On the other hand, we wanted to *allow* object-oriented programming with Lua. Instead of fixing a model, we decided to provide flexible mechanisms that would allow the programmer to build whatever model was suitable to the application. Lua 2.1, released in February 1995, marked the introduction of these *extensible semantics* mechanisms, which have greatly increased the expressiveness of Lua. Extensible semantics has become a hallmark of Lua.

One of the goals of extensible semantics was to allow tables to be used as a basis for objects and classes. For that, we needed to implement inheritance for tables. Another goal was to turn userdata into natural proxies for application data, not merely handles meant to be used solely as arguments to functions. We wanted to be able to index userdata as if they were tables and to call methods on them. This would allow Lua to fulfill one of its main design goals more naturally: to extend applications by providing scriptable access to application services and data. Instead of adding mechanisms to support all these features directly in the language, we decided that it would be conceptually simpler to define a more

general *fallback* mechanism to let the programmer intervene whenever Lua did not know how to proceed.

We introduced fallbacks in Lua 2.1 and defined them for the following operations: table indexing, arithmetic operations, string concatenation, order comparisons, and function calls.[3] When one of these operations was applied to the "wrong" kind of values, the corresponding fallback was called, allowing the programmer to determine how Lua would proceed. The table indexing fallbacks allowed userdata (and other values) to behave as tables, which was one of our motivations. We also defined a fallback to be called when a key was absent from a table, so that we could support many forms of inheritance (through delegation). To complete the support for object-oriented programming, we added two pieces of syntactic sugar: method definitions of the form 'function a:foo(···)' as sugar for 'function a.foo(self,···)' and method calls of the form 'a:foo(···)' as sugar for 'a.foo(a,···)'. In §6.8 we discuss fallbacks in detail and how they evolved into their later incarnations: tag methods and metamethods.

Since Lua 1.0, we have provided introspective functions for values: type, which queries the type of a Lua value; next, which traverses a table; and nextvar, which traverses the global environment. (As mentioned in §4, this was partially motivated by the need to implement SOL-like type checking.) In response to user pressure for full debug facilities, Lua 2.2 (November 1995) introduced a debug API to provide information about running functions. This API gave users the means to write in C their own introspective tools, such as debuggers and profilers. The debug API was initially quite simple: it allowed access to the Lua call stack, to the currently executing line, and provided a function to find the name of a variable holding a given value. Following the M.Sc. work of Tomás Gorham [22], the debug API was improved in Lua 2.4 (May 1996) by functions to access local variables and hooks to be called at line changes and function calls.

With the widespread use of Lua at Tecgraf, many large graphics metafiles were being written in Lua as the output of graphical editors. Loading such metafiles was taking increasingly longer as they became larger and more complex.[4] Since its first version, Lua precompiled all programs to bytecode just before running them. The load time of a large program could be substantially reduced by saving this bytecode to a file. This would be especially relevant for procedural data files such as graphics metafiles. So, in Lua 2.4, we introduced an external compiler, called luac, which precompiled a Lua program and saved the generated bytecode to a binary file. (Our first paper about Lua [15] had already an-

ticipated the possibility of an external compiler.) The format of this file was chosen to be easily loaded and reasonably portable. With luac, programmers could avoid parsing and code generation at run time, which in the early days were costly. Besides faster loading, luac also allowed off-line syntax checking and protection from casual user changes. Many products (e.g., The Sims and Adobe Lightroom) distribute Lua scripts in precompiled form.

During the implementation of luac, we started to restructure Lua's core into clearly separated modules. As a consequence, it is now quite easy to remove the parsing modules (lexer, parser, and code generator), which currently represent 35% of the core code, leaving just the module that loads precompiled Lua programs, which is merely 3% of the core code. This reduction can be significant when embedding Lua in small devices such as mobile devices, robots and sensors.[5]

Since its first version, Lua has included a library for string-processing. The facilities provided by this library were minimal until Lua 2.4. However, as Lua matured, it became desirable to do heavier text processing in Lua. We thought that a natural addition to Lua would be pattern matching, in the tradition of Snobol, Icon, Awk, and Perl. However, we did not want to include a third-party pattern-matching engine in Lua because such engines tend to be very large; we also wanted to avoid copyright issues that could be raised by including third-party code in Lua.

As a student project supervised by Roberto in the second semester of 1995, Milton Jonathan, Pedro Miller Rabinovitch, Pedro Willemsens, and Vinicius Almendra produced a pattern-matching library for Lua. Experience with that design led us to write our own pattern-matching engine for Lua, which we added to Lua 2.5 (November 1996) in two functions: strfind (which originally only found plain substrings) and the new gsub function (a name taken from Awk). The gsub function globally replaced substrings matching a given pattern in a larger string. It accepted either a replacement string or a function that was called each time a match was found and was intended to return the replacement string for that match. (That was an innovation at the time.) Aiming at a small implementation, we did not include full regular expressions. Instead, the patterns understood by our engine were based on character classes, repetitions, and captures (but not alternation or grouping). Despite its simplicity, this kind of pattern matching is quite powerful and was an important addition to Lua.

That year was a turning point in the history of Lua because it gained international exposure. In June 1996 we published a paper about Lua in *Software: Practice & Experience* [30] that brought external attention to Lua, at least in

---

[3] We also introduced fallbacks for handling fatal errors and for monitoring garbage collection, even though they were not part of extensible semantics.

[4] Surprisingly, a substantial fraction of the load time was taken in the lexer for converting real numbers from text form to floating-point representation. Real numbers abound in graphics metafiles.

[5] Crazy Ivan, a robot that won RoboCup in 2000 and 2001 in Denmark, had a "brain" implemented in Lua. It ran directly on a Motorola Coldfire 5206e processor without any operating system (in other words, Lua was the operating system). Lua was stored on a system ROM and loaded programs at startup from the serial port.

academic circles.[6] In December 1996, shortly after Lua 2.5 was released, the magazine *Dr. Dobb's Journal* featured an article about Lua [16]. *Dr. Dobb's Journal* is a popular publication aimed directly at programmers, and that article brought Lua to the attention of the software industry. Among several messages that we received right after that publication was one sent in January 1997 by Bret Mogilefsky, who was the lead programmer of Grim Fandango, an adventure game then under development by LucasArts. Bret told us that he had read about Lua in *Dr. Dobb's* and that they planned to replace their home-brewed scripting language with Lua. Grim Fandango was released in October 1998 and in May 1999 Bret told us that "a *tremendous* amount of the game was written in Lua" (his emphasis) [38].[7] Around that time, Bret attended a roundtable about game scripting at the Game Developers' Conference (GDC, the main event for game programmers) and at the end he related his experience with the successful use of Lua in Grim Fandango. We know of several developers who first learned about Lua at that event. After that, Lua spread by word of mouth among game developers to become a definitely marketable skill in the game industry (see §8).

As a consequence of Lua's international exposure, the number of messages sent to us asking questions about Lua increased substantially. To handle this traffic more efficiently, and also to start building a Lua community, so that other people could answer Lua questions, in February 1997 we created a mailing list for discussing Lua. Over 38,000 messages have been posted to this list since then. The use of Lua in many popular games has attracted many people to the list, which now has over 1200 subscribers. We have been fortunate that the Lua list is very friendly and at the same time very technical. The list has become the focal point of the Lua community and has been a source of motivation for improving Lua. All important events occur first in the mailing list: release announcements, feature requests, bug reports, etc.

The creation of a `comp.lang.lua` Usenet newsgroup was discussed twice in the list over all these years, in April 1998 and in July 1999. The conclusion both times was that the traffic in the list did not warrant the creation of a newsgroup. Moreover, most people preferred a mailing list. The creation of a newsgroup seems no longer relevant because there are several web interfaces for reading and searching the complete list archives.

---

[6] In November 1997, that article won the First Prize (technological category) in the II Compaq Award for Research and Development in Computer Science, a joint venture of Compaq Computer in Brazil, the Brazilian Ministry of Science and Technology, and the Brazilian Academy of Sciences.

[7] Grim Fandango mentioned Lua and PUC-Rio in its final credits. Several people at PUC-Rio first learned about Lua from that credit screen, and were surprised to learn that Brazilian software was part of a hit game. It has always bothered us that Lua is widely known abroad but has remained relatively unknown in Brazil until quite recently.

5.3   Lua 3

The fallback mechanism introduced in Lua 2.1 to support extensible semantics worked quite well but it was a global mechanism: there was only one hook for each event. This made it difficult to share or reuse code because modules that defined fallbacks for the same event could not co-exist easily. Following a suggestion by Stephan Herrmann in December 1996, in Lua 3.0 (July 1997) we solved the fallback clash problem by replacing fallbacks with *tag methods*: the hooks were attached to pairs (*event*, *tag*) instead of just to events. Tags had been introduced in Lua 2.1 as integer labels that could be attached to userdata (see §6.10); the intention was that C objects of the same type would be represented in Lua by userdata having the same tag. (However, Lua did not force any interpretation on tags.) In Lua 3.0 we extended tags to all values to support tag methods. The evolution of fallbacks is discussed in §6.8.

Lua 3.1 (July 1998) brought functional programming to Lua by introducing anonymous functions and function closures via "upvalues". (Full lexical scoping had to wait until Lua 5.0; see §6.6.) The introduction of closures was mainly motivated by the existence of higher-order functions, such as `gsub`, which took functions as arguments. During the work on Lua 3.1, there were discussions in the mailing list about multithreading and cooperative multitasking, mainly motivated by the changes Bret Mogilefsky had made to Lua 2.5 and 3.1 alpha for Grim Fandango. No conclusions were reached, but the topic remained popular. Cooperative multitasking in Lua was finally provided in Lua 5.0 (April 2003); see §6.7.

The C API remained largely unchanged from Lua 1.0 to Lua 3.2; it worked over an implicit Lua state. However, newer applications, such as web services, needed multiple states. To mitigate this problem, Lua 3.1 introduced multiple independent Lua states that could be switched at run time. A fully reentrant API would have to wait until Lua 4.0. In the meantime, two unofficial versions of Lua 3.2 with explicit Lua states appeared: one written in 1998 by Roberto Ierusalimschy and Anna Hester based on Lua 3.2 alpha for CGILua [26], and one written in 1999 by Erik Hougaard based on Lua 3.2 final. Erik's version was publicly available and was used in the Crazy Ivan robot. The version for CGILua was released only as part of the CGILua distribution; it never existed as an independent package.

Lua 3.2 (July 1999) itself was mainly a maintenance release; it brought no novelties except for a debug library that allowed tools to be written in Lua instead of C. Nevertheless, Lua was quite stable by then and Lua 3.2 had a long life. Because the next version (Lua 4.0) introduced a new, incompatible API, many users just stayed with Lua 3.2 and never migrated to Lua 4.0. For instance, Tecgraf never migrated to Lua 4.0, opting to move directly to Lua 5.0; many products at Tecgraf still use Lua 3.2.

5.4 Lua 4

Lua 4.0 was released in November 2000. As mentioned above, the main change in Lua 4.0 was a fully reentrant API, motivated by applications that needed multiple Lua states. Since making the API fully reentrant was already a major change, we took the opportunity and completely redesigned the API around a clear stack metaphor for exchanging values with C (see §6.9). This was first suggested by Reuben Thomas in July 2000.

Lua 4.0 also introduced a 'for' statement, then a top item in the wish-list of most Lua users and a frequent topic in the mailing list. We had not included a 'for' statement earlier because 'while' loops were more general. However, users complained that they kept forgetting to update the control variable at the end of 'while' loops, thus leading to infinite loops. Also, we could not agree on a good syntax. We considered the Modula 'for' too restrictive because it did not cover iterations over the elements of a table or over the lines of a file. A 'for' loop in the C tradition did not fit with the rest of Lua. With the introduction of closures and anonymous functions in Lua 3.1, we decided to use higher-order functions for implementing iterations. So, Lua 3.1 provided a higher-order function that iterated over a table by calling a user-supplied function over all pairs in the table. To print all pairs in a table t, one simply said 'foreach(t,print)'.

In Lua 4.0 we finally designed a 'for' loop, in two variants: a numeric loop and a table-traversal loop (first suggested by Michael Spalinski in October 1997). These two variants covered most common loops; for a really generic loop, there was still the 'while' loop. Printing all pairs in a table t could then be done as follows:[8]

```
for k,v in t do
  print(k,v)
end
```

The addition of a 'for' statement was a simple one but it did change the look of Lua programs. In particular, Roberto had to rewrite many examples in his draft book on Lua programming. Roberto had been writing this book since 1998, but he could never finish it because Lua was a moving target. With the release of Lua 4.0, large parts of the book and almost all its code snippets had to be rewritten.

Soon after the release of Lua 4.0, we started working on Lua 4.1. Probably the main issue we faced for Lua 4.1 was whether and how to support multithreading, a big issue at that time. With the growing popularity of Java and Pthreads, many programmers began to consider support for multithreading as an essential feature in any programming language. However, for us, supporting multithreading in Lua posed serious questions. First, to implement multithreading in C requires primitives that are not part of ANSI C —

although Pthreads was popular, there were (and still there are) many platforms without this library. Second, and more important, we did not (and still do not) believe in the standard multithreading model, which is preemptive concurrency with shared memory: we still think that no one can write correct programs in a language where 'a=a+1' is not deterministic.

For Lua 4.1, we tried to solve those difficulties in a typical Lua fashion: we implemented only a basic mechanism of multiple stacks, which we called *threads*. External libraries could use those Lua threads to implement multithreading, based on a support library such as Pthreads. The same mechanism could be used to implement coroutines, in the form of non-preemptive, collaborative multithreading. Lua 4.1 alpha was released in July 2001 with support for external multithreading and coroutines; it also introduced support for weak tables and featured a register-based virtual machine, with which we wanted to experiment.

The day after Lua 4.1 alpha was released, John D. Ramsdell started a big discussion in the mailing list about lexical scoping. After several dozen messages, it became clear that Lua needed full lexical scoping, instead of the upvalue mechanism adopted since Lua 3.1. By October 2001 we had come up with an efficient implementation of full lexical scoping, which we released as a work version in November 2001. (See §6.6 for a detailed discussion of lexical scoping.) That version also introduced a new hybrid representation for tables that let them be implemented as arrays when appropriate (see §6.2 for further details). Because that version implemented new basic algorithms, we decided to release it as a work version, even though we had already released an alpha version for Lua 4.1.

In February 2002 we released a new work version for Lua 4.1, with three relevant novelties: a generic 'for' loop based on iterator functions, metatables and metamethods as a replacement for tags and fallbacks[9] (see §6.8), and coroutines (see §6.7). After that release, we realized that Lua 4.1 would bring too many major changes — perhaps 'Lua 5.0' would be a better name for the next version.

5.5 Lua 5

The final blow to the name 'Lua 4.1' came a few days later, during the Lua Library Design Workshop organized by Christian Lindig and Norman Ramsey at Harvard. One of the main conclusions of the workshop was that Lua needed some kind of module system. Although we had always considered that modules could be implemented using tables, not even the standard Lua libraries followed this path. We then decided to take that step for the next version.

---

[8] With the introduction of 'for' iterators in Lua 5.0, this syntax was marked as obsolete and later removed in Lua 5.1.

[9] The use of ordinary Lua tables for implementing extensible semantics had already been suggested by Stephan Herrmann in December 1996, but we forgot all about it until it was suggested again by Edgar Toernig in October 2000, as part of a larger proposal, which he called 'unified methods'. The term 'metatable' was suggested by Rici Lake in November 2001.

Packaging library functions inside tables had a big practical impact, because it affected any program that used at least one library function. For instance, the old `strfind` function was now called `string.find` (field 'find' in string library stored in the 'string' table); `openfile` became `io.open`; `sin` became `math.sin`; and so on. To make the transition easier, we provided a compatibility script that defined the old functions in terms of the new ones:

```
strfind = string.find
openfile = io.open
sin = math.sin

...
```

Nevertheless, packaging libraries in tables was a major change. In June 2002, when we released the next work version incorporating this change, we dropped the name 'Lua 4.1' and named it 'Lua 5.0 work0'. Progress to the final version was steady from then on and Lua 5.0 was released in April 2003. This release froze Lua enough to allow Roberto to finish his book, which was published in December 2003 [27].

Soon after the release of Lua 5.0 we started working on Lua 5.1. The initial motivation was the implementation of incremental garbage collection in response to requests from game developers. Lua uses a traditional mark-and-sweep garbage collector, and, until Lua 5.0, garbage collection was performed atomically. As a consequence, some applications might experience potentially long pauses during garbage collection.[10] At that time, our main concern was that adding the write barriers needed to implement an incremental garbage collector would have a negative impact on Lua performance. To compensate for that we tried to make the collector generational as well. We also wanted to keep the adaptive behavior of the old collector, which adjusted the frequency of collection cycles according to the total memory in use. Moreover, we wanted to keep the collector simple, like the rest of Lua.

We worked on the incremental generational garbage collector for over a year. But since we did not have access to applications with strong memory requirements (like games), it was difficult for us to test the collector in real scenarios. From March to December 2004 we released several work versions trying to get concrete feedback on the performance of the collector in real applications. We finally received reports of bizarre memory-allocation behavior, which we later managed to reproduce but not explain. In January 2005, Mike Pall, an active member of the Lua community, came up with memory-allocation graphs that explained the problem: in some scenarios, there were subtle interactions between the incremental behavior, the generational behavior, and the adaptive behavior, such that the collector "adapted"

for less and less frequent collections. Because it was getting too complicated and unpredictable, we gave up the generational aspect and implemented a simpler incremental collector in Lua 5.1.

During that time, programmers had been experimenting with the module system introduced in Lua 5.0. New packages started to be produced, and old packages migrated to the new system. Package writers wanted to know the best way to build modules. In July 2005, during the development of Lua 5.1, an international Lua workshop organized by Mark Hamburg was held at Adobe in San Jose. (A similar workshop organized by Wim Couwenberg and Daniel Silverstone was held in September 2006 at Océ in Venlo.) One of the presentations was about the novelties of Lua 5.1, and there were long discussions about modules and packages. As a result, we made a few small but significant changes in the module system. Despite our "mechanisms, not policy" guideline for Lua, we defined a set of policies for writing modules and loading packages, and made small changes to support these policies better. Lua 5.1 was released in February 2006. Although the original motivation for Lua 5.1 was incremental garbage collection, the improvement in the module system was probably the most visible change. On the other hand, that incremental garbage collection remained invisible shows that it succeeded in avoiding long pauses.

6. Feature evolution

In this section, we discuss in detail the evolution of some of the features of Lua.

6.1 Types

Types in Lua have been fairly stable. For a long time, Lua had only six basic types: nil, number, string, table, function, and userdata. (Actually, until Lua 3.0, C functions and Lua functions had different types internally, but that difference was transparent to callers.) The only real change happened in Lua 5.0, which introduced two new types: threads and booleans.

The type thread was introduced to represent coroutines. Like all other Lua values, threads are first-class values. To avoid creating new syntax, all primitive operations on threads are provided by a library.

For a long time we resisted introducing boolean values in Lua: nil was false and anything else was true. This state of affairs was simple and seemed sufficient for our purposes. However, nil was also used for absent fields in tables and for undefined variables. In some applications, it is important to allow table fields to be marked as false but still be seen as present; an explicit false value can be used for this. In Lua 5.0 we finally introduced boolean values *true* and *false*. Nil is still treated as false. In retrospect, it would probably have been better if nil raised an error in boolean expressions, as it does in other expressions. This would be more consistent with its role as proxy for undefined values. How-

---

[10] Erik Hougaard reported that the Crazy Ivan robot would initially drive off course when Lua performed garbage collection (which could take a half second, but that was enough). To stay in course, they had to stop both motors and pause the robot during garbage collection.

ever, such a change would probably break many existing programs. LISP has similar problems, with the empty list representing both nil and false. Scheme explicitly represents false and treats the empty list as true, but some implementations of Scheme still treat the empty list as false.

6.2 Tables

Lua 1.1 had three syntactical constructs to create tables: '@()', '@[]', and '@{}'. The simplest form was '@()', which created an empty table. An optional size could be given at creation time, as an efficiency hint. The form '@[]' was used to create arrays, as in '@[2,4,9,16,25]'. In such tables, the keys were implicit natural numbers starting at 1. The form '@{}' was used to create records, as in '@{name="John",age=35}'. Such tables were sets of key-value pairs in which the keys were explicit strings. A table created with any of those forms could be modified dynamically after creation, regardless of how it had been created. Moreover, it was possible to provide user functions when creating lists and records, as in '@foo[]' or '@foo{}'. This syntax was inherited from SOL and was the expression of procedural data description, a major feature of Lua (see §2). The semantics was that a table was created and then the function was called with that table as its single argument. The function was allowed to check and modify the table at will, but its return values were ignored: the table was the final value of the expression.

In Lua 2.1, the syntax for table creation was unified and simplified: the leading '@' was removed and the only constructor became '{···}'. Lua 2.1 also allowed mixed constructors, such as

```
grades{8.5, 6.0, 9.2; name="John", major="math"}
```

in which the array part was separated from the record part by a semicolon. Finally, 'foo{···}' became sugar for 'foo({···})'. In other words, table constructors with functions became ordinary function calls. As a consequence, the function had to explicitly return the table (or whatever value it chose). Dropping the '@' from constructors was a trivial change, but it actually changed the feel of the language, not merely its looks. Trivial changes that improve the feel of a language are not to be overlooked.

This simplification in the syntax and semantics of table constructors had a side-effect, however. In Lua 1.1, the equality operator was '='. With the unification of table constructors in Lua 2.1, an expression like '{a=3}' became ambiguous, because it could mean a table with either a pair ("a", 3) or a pair $(1, b)$, where b is the value of the equality 'a=3'. To solve this ambiguity, in Lua 2.1 we changed the equality operator from '=' to '=='. With this change, '{a=3}' meant a table with the pair ("a", 3), while '{a==3}' meant a table with the pair $(1, b)$.

These changes made Lua 2.1 incompatible with Lua 1.1 (hence the change in the major version number). Nevertheless, since at that time virtually all Lua users were from Tec-

graf, this was not a fatal move: existing programs were easily converted with the aid of ad-hoc tools that we wrote for this task.

The syntax for table constructors has since remained mostly unchanged, except for an addition introduced in Lua 3.1: keys in the record part could be given by any expression, by enclosing the expression inside brackets, as in '{[10*x+f(y)]=47}'. In particular, this allowed keys to be arbitrary strings, including reserved words and strings with spaces. Thus, '{function=1}' is not valid (because 'function' is a reserved word), but '{["function"]=1}' is valid. Since Lua 5.0, it is also possible to freely intermix the array part and the record part, and there is no need to use semicolons in table constructors.

While the syntax of tables has evolved, the semantics of tables in Lua has not changed at all: tables are still associative arrays and can store arbitrary pairs of values. However, frequently in practice tables are used solely as arrays (that is, with consecutive integer keys) or solely as records (that is, with string keys). Because tables are the only data-structuring mechanism in Lua, we have invested much effort in implementing them efficiently inside Lua's core. Until Lua 4.0, tables were implemented as pure hash tables, with all pairs stored explicitly. In Lua 5.0 we introduced a hybrid representation for tables: every table contains a hash part and an array part, and both parts can be empty. Lua detects whether a table is being used as an array and automatically stores the values associated to integer indices in the array part, instead of adding them to the hash part [31]. This division occurs only at a low implementation level; access to table fields is transparent, even to the virtual machine. Tables automatically adapt their two parts according to their contents.

This hybrid scheme has two advantages. First, access to values with integer keys is faster because no hashing is needed. Second, and more important, the array part takes roughly half the memory it would take if it were stored in the hash part, because the keys are implicit in the array part but explicit in the hash part. As a consequence, if a table is being used as an array, it performs as an array, as long as its integer keys are densely distributed. Moreover, no memory or time penalty is paid for the hash part, because it does not even exist. Conversely, if the table is being used as a record and not as an array, then the array part is likely to be empty. These memory savings are important because it is common for a Lua program to create many small tables (e.g., when tables are used to represent objects). Lua tables also handle sparse arrays gracefully: the statement 'a={[1000000000]=1}' creates a table with a single entry in its hash part, not an array with one billion elements.

Another reason for investing effort into an efficient implementation of tables is that we can use tables for all kinds of tasks. For instance, in Lua 5.0 the standard library functions, which had existed since Lua 1.1 as global variables,

were moved to fields inside tables (see §5.5). More recently, Lua 5.1 brought a complete package and module system based on tables.

Tables play a prominent role in Lua's core. On two occasions we have been able to replace special data structures inside the core with ordinary Lua tables: in Lua 4.0 for representing the global environment (which keeps all global variables) and in Lua 5.0 for implementing extensible semantics (see §6.8). Starting with Lua 4.0, global variables are stored in an ordinary Lua table, called the table of globals, a simplification suggested by John Belmonte in April 2000. In Lua 5.0 we replaced tags and tag methods (introduced in Lua 3.0) by metatables and metamethods. Metatables are ordinary Lua tables and metamethods are stored as fields in metatables. Lua 5.0 also introduced environment tables that can be attached to Lua functions; they are the tables where global names in Lua functions are resolved at run time. Lua 5.1 extended environment tables to C functions, userdata, and threads, thus replacing the notion of global environment. These changes simplified both the implementation of Lua and the API for Lua and C programmers, because globals and metamethods can be manipulated within Lua without the need for special functions.

6.3 Strings

Strings play a major role in scripting languages and so the facilities to create and manipulate strings are an important part of the usability of such languages.

The syntax for literal strings in Lua has had an interesting evolution. Since Lua 1.1, a literal string can be delimited by matching single or double quotes, and can contain C-like escape sequences. The use of both single and double quotes to delimit strings with the same semantics was a bit unusual at the time. (For instance, in the tradition of shell languages, Perl expands variables inside double-quoted strings, but not inside single-quoted strings.) While these dual quotes allow strings to contain one kind of quote without having to escape it, escape sequences are still needed for arbitrary text.

Lua 2.2 introduced *long strings*, a feature not present in classical programming languages, but present in most scripting languages.[11] Long strings can run for several lines and do not interpret escape sequences; they provide a convenient way to include arbitrary text as a string, without having to worry about its contents. However, it is not trivial to design a good syntax for long strings, especially because it is common to use them to include arbitrary program text (which may contain other long strings). This raises the question of how long strings end and whether they may nest. Until Lua 5.0, long strings were wrapped inside matching '[[···]]' and could contain nested long strings. Unfortunately, the closing delimiter ']]' could easily be part of a valid Lua program in an unbalanced way, as in 'a[b[i]]',

or in other contexts, such as '<![CDATA[···]]>' from XML. So, it was hard to reliably wrap arbitrary text as a long string.

Lua 5.1 introduced a new form for long strings: text delimited by matching '[===[···]===]', where the number of '=' characters is arbitrary (including zero). These new long strings do not nest: a long string ends as soon as a closing delimiter with the right number of '=' is seen. Nevertheless, it is now easy to wrap arbitrary text, even text containing other long strings or unbalanced ']=···=]' sequences: simply use an adequate number of '=' characters.

6.4 Block comments

Comments in Lua are signaled by '--' and continue to the end of the line. This is the simplest kind of comment, and is very effective. Several other languages use single-line comments, with different marks. Languages that use '--' for comments include Ada and Haskell.

We never felt the need for multi-line comments, or block comments, except as a quick way to disable code. There was always the question of which syntax to use: the familiar '/*···*/' syntax used in C and several other languages does not mesh well with Lua's single-line comments. There was also the question of whether block comments could nest or not, always a source of noise for users and of complexity for the lexer. Nested block comments happen when programmers want to 'comment out' some block of code, to disable it. Naturally, they expect that comments inside the block of code are handled correctly, which can only happen if block comments can be nested.

ANSI C supports block comments but does not allow nesting. C programmers typically disable code by using the C preprocessor idiom '#if 0 ··· #endif'. This scheme has the clear advantage that it interacts gracefully with existing comments in the disabled code. With this motivation and inspiration, we addressed the need for disabling blocks of code in Lua — not the need for block comments — by introducing conditional compilation in Lua 3.0 via pragmas inspired in the C preprocessor. Although conditional compilation could be used for block comments, we do not think that it ever was. During work on Lua 4.0, we decided that the support for conditional compilation was not worth the complexity in the lexer and in its semantics for the user, especially after not having reached any consensus about a full macro facility (see §7). So, in Lua 4.0 we removed support for conditional compilation and Lua remained without support for block comments.[12]

Block comments were finally introduced in Lua 5.0, in the form '--[[···]]'. Because they intentionally mimicked the syntax of long strings (see §6.3), it was easy to modify the lexer to support block comments. This similarity also helped users to grasp both concepts and their syntax. Block

---

[11] 'Long string' is a Lua term. Other languages use terms such as 'verbatim text' or 'heredoc'.

[12] A further motivation was that by that time we had found a better way to generate and use debug information, and so the pragmas that controlled this were no longer needed. Removing conditional compilation allowed us to get rid of all pragmas.

comments can also be used to disable code: the idiom is to surround the code between two lines containing '`--[[`' and '`--]]`'. The code inside those lines can be re-enabled by simply adding a single '`-`' at the start of the first line: both lines then become harmless single-line comments.

Like long strings, block comments could nest, but they had the same problems as long strings. In particular, valid Lua code containing unbalanced '`]]`'s, such as '`a[b[i]]`', could not be reliably commented out in Lua 5.0. The new scheme for long strings in Lua 5.1 also applies to block comments, in the form of matching '`--[===[···]===]`', and so provides a simple and robust solution for this problem.

6.5 Functions

Functions in Lua have always been first-class values. A function can be created at run time by compiling and executing a string containing its definition.[13] Since the introduction of anonymous functions and upvalues in Lua 3.1, programmers are able to create functions at run time without resorting to compilation from text.

Functions in Lua, whether written in C or in Lua, have no declaration. At call time they accept a variable number of arguments: excess arguments are discarded and missing arguments are given the value nil. (This coincides with the semantics of multiple assignment.) C functions have always been able to handle a variable number of arguments. Lua 2.5 introduced *vararg* Lua functions, marked by a parameter list ending in '`...`' (an experimental feature that became official only in Lua 3.0). When a vararg function was called, the arguments corresponding to the dots were collected into a table named '`arg`'. While this was simple and mostly convenient, there was no way to pass those arguments to another function, except by unpacking this table. Because programmers frequently want to just pass the arguments along to other functions, Lua 5.1 allows '`...`' to be used in argument lists and on the right-hand side of assignments. This avoids the creation of the '`arg`' table if it is not needed.

The unit of execution of Lua is called a *chunk*; it is simply a sequence of statements. A chunk in Lua is like the main program in other languages: it can contain both function definitions and executable code. (Actually, a function definition is executable code: an assignment.) At the same time, a chunk closely resembles an ordinary Lua function. For instance, chunks have always had exactly the same kind of bytecode as ordinary Lua functions. However, before Lua 5.0, chunks needed some internal magic to start executing. Chunks began to look like ordinary functions in Lua 2.2, when local variables outside functions were allowed as an undocumented feature (that became official only in Lua 3.1). Lua 2.5 allowed chunks to return values. In Lua 3.0 chunks became functions internally, except that they were executed

right after being compiled; they did not exist as functions at the user level. This final step was taken in Lua 5.0, which broke the loading and execution of chunks into two steps, to provide host programmers better control for handling and reporting errors. As a consequence, in Lua 5.0 chunks became ordinary anonymous functions with no arguments. In Lua 5.1 chunks became anonymous vararg functions and thus can be passed values at execution time. Those values are accessed via the new '`...`' mechanism.

From a different point of view, chunks are like modules in other languages: they usually provide functions and variables to the global environment. Originally, we did not intend Lua to be used for large-scale programming and so we did not feel the need to add an explicit notion of modules to Lua. Moreover, we felt that tables would be sufficient for building modules, if necessary. In Lua 5.0 we made that feeling explicit by packaging all standard libraries into tables. This encouraged other people to do the same and made it easier to share libraries. We now feel that Lua can be used for large-scale programming, especially after Lua 5.1 brought a package system and a module system, both based on tables.

6.6 Lexical scoping

From an early stage in the development of Lua we started thinking about first-class functions with full lexical scoping. This is an elegant construct that fits well within Lua's philosophy of providing few but powerful constructs. It also makes Lua apt for functional programming. However, we could not figure out a reasonable implementation for full lexical scoping. Since the beginning Lua has used a simple array stack to keep activation records (where all local variables and temporaries live). This implementation had proved simple and efficient, and we saw no reason to change it. When we allow nested functions with full lexical scoping, a variable used by an inner function may outlive the function that created it, and so we cannot use a stack discipline for such variables.

Simple Scheme implementations allocate frames in the heap. Already in 1987, Dybvig [20] described how to use a stack to allocate frames, provided that those frames did not contain variables used by nested functions. His method requires that the compiler know beforehand whether a variable appears as a free variable in a nested function. This does not suit the Lua compiler because it generates code to manipulate variables as soon as it parses an expression; at that moment, it cannot know whether any variable is later used free in a nested function. We wanted to keep this design for implementing Lua, because of its simplicity and efficiency, and so could not use Dybvig's method. For the same reason, we could not use advanced compiler techniques, such as data-flow analysis.

Currently there are several optimization strategies to avoid using the heap for frames (e.g., [21]), but they all need compilers with intermediate representations, which the Lua compiler does not use. McDermott's proposal for stack frame allocation [36], which is explicitly addressed to inter-

---

[13] Some people maintain that the ability to evaluate code from text at run time and within the environment of the running program is what characterizes scripting languages.

preters, is the only one we know of that does not require intermediate representation for code generation. Like our current implementation [31], his proposal puts variables in the stack and moves them to the heap on demand, if they go out of scope while being used by a nested closure. However, his proposal assumes that environments are represented by association lists. So, after moving an environment to the heap, the interpreter has to correct only the list header, and all accesses to local variables automatically go to the heap. Lua uses real records as activation records, with local-variable access being translated to direct accesses to the stack plus an offset, and so cannot use McDermott's method.

For a long time those difficulties kept us from introducing nested first-class functions with full lexical scoping in Lua. Finally, in Lua 3.1 we settled on a compromise that we called *upvalues*. In this scheme, an inner function cannot access and modify external variables when it runs, but it can access the values those variables had when the function was created. Those values are called *upvalues*. The main advantage of upvalues is that they can be implemented with a simple scheme: all local variables live in the stack; when a function is created, it is wrapped in a closure containing copies of the values of the external variables used by the function. In other words, upvalues are the frozen values of external variables.[14] To avoid misunderstandings, we created a new syntax for accessing upvalues: '`%varname`'. This syntax made it clear that the code was accessing the frozen value of that variable, not the variable itself. Upvalues proved to be very useful, despite being immutable. When necessary, we could simulate mutable external variables by using a table as the upvalue: although we could not change the table itself, we could change its fields. This feature was especially useful for anonymous functions passed to higher-order functions used for table traversal and pattern matching.

In December 2000, Roberto wrote in the first draft of his book [27] that "Lua has a form of proper lexical scoping through upvalues." In July 2001 John D. Ramsdell argued in the mailing list that "a language is either lexically scoped or it is not; adding the adjective 'proper' to the phrase 'lexical scoping' is meaningless." That message stirred us to search for a better solution and a way to implement full lexical scoping. By October 2001 we had an initial implementation of full lexical scoping and described it to the list. The idea was to access each upvalue through an indirection that pointed to the stack while the variable was in scope; at the end of the scope a special virtual machine instruction "closed" the upvalue, moving the variable's value to a heap-allocated space and correcting the indirection to point there. Open closures (those with upvalues still pointing to the stack) were kept in a list to allow their correction and

the reuse of open upvalues. Reuse is essential to get the correct semantics. If two closures, sharing an external variable, have their own upvalues, then at the end of the scope each closure will have its own copy of the variable, but the correct semantics dictates that they should share the variable. To ensure reuse, the algorithm that created closures worked as follows: for each external variable used by the closure, it first searched the list of open closures. If it found an upvalue pointing to that external variable, it reused that upvalue; otherwise, it created a new upvalue.

Edgar Toering, an active member of the Lua community, misunderstood our description of lexical scoping. It turned out that the way he understood it was better than our original idea: instead of keeping a list of open closures, keep a list of open upvalues. Because the number of local variables used by closures is usually smaller than the number of closures using them (the first is statically limited by the program text), his solution was more efficient than ours. It was also easier to adapt to coroutines (which were being implemented at around the same time), because we could keep a separate list of upvalues for each stack. We added full lexical scoping to Lua 5.0 using this algorithm because it met all our requirements: it could be implemented with a one-pass compiler; it imposed no burden on functions that did not access external local variables, because they continued to manipulate all their local variables in the stack; and the cost to access an external local variable was only one extra indirection [31].

6.7 Coroutines

For a long time we searched for some kind of first-class continuations for Lua. This search was motivated by the existence of first-class continuations in Scheme (always a source of inspiration to us) and by demands from game programmers for some mechanism for "soft" multithreading (usually described as "some way to suspend a character and continue it later").

In 2000, Maria Julia de Lima implemented full first-class continuations on top of Lua 4.0 alpha, as part of her Ph.D. work [35]. She used a simple approach because, like lexical scoping, smarter techniques to implement continuations were too complex compared to the overall simplicity of Lua. The result was satisfactory for her experiments, but too slow to be incorporated in a final product. Nevertheless, her implementation uncovered a problem peculiar to Lua. Since Lua is an extensible extension language, it is possible (and common) to call Lua from C and C from Lua. Therefore, at any given point in the execution of a Lua program, the current continuation usually has parts in Lua mixed with parts in C. Although it is possible to manipulate a Lua continuation (essentially by manipulating the Lua call stack), it is impossible to manipulate a C continuation within ANSI C. At that time, we did not understand this problem deeply enough. In particular, we could not figure out what the exact restrictions related to C calls were. Lima simply forbade any C calls in her implementation. Again, that solution was

---

[14] A year later Java adopted a similar solution to allow inner classes. Instead of freezing the value of an external variable, Java insists that you can only access *final* variables in inner classes, and so ensures that the variable is frozen.

satisfactory for her experiments, but unacceptable for an official Lua version because the ease of mixing Lua code with C code is one of Lua's hallmarks.

Unaware of this difficulty, in December 2001 Thatcher Ulrich announced in the mailing list:

> I've created a patch for Lua 4.0 that makes calls from Lua to Lua non-recursive (i.e., 'stackless'). This allows the implementation of a 'sleep()' call, which exits from the host program [. . .], and leaves the Lua state in a condition where the script can be resumed later via a call to a new API function, lua_resume.

In other words, he proposed an asymmetric coroutine mechanism, based on two primitives: *yield* (which he called *sleep*) and *resume*. His patch followed the high-level description given in the mailing list by Bret Mogilefsky on the changes made to Lua 2.5 and 3.1 to add cooperative multitasking in Grim Fandango. (Bret could not provide details, which were proprietary.)

Shortly after this announcement, during the Lua Library Design Workshop held at Harvard in February 2002, there was some discussion about first-class continuations in Lua. Some people claimed that, if first-class continuations were deemed too complex, we could implement one-shot continuations. Others argued that it would be better to implement symmetric coroutines. But we could not find a proper implementation of any of these mechanisms that could solve the difficulty related to C calls.

It took us some time to realize why it was hard to implement symmetric coroutines in Lua, and also to understand how Ulrich's proposal, based on asymmetric coroutines, avoided our difficulties. Both one-shot continuations and symmetric coroutines involve the manipulation of full continuations. So, as long as these continuations include any C part, it is impossible to capture them (except by using facilities outside ANSI C). In contrast, an asymmetric coroutine mechanism based on *yield* and *resume* manipulates *partial* continuations: *yield* captures the continuation up to the corresponding *resume* [19]. With asymmetric coroutines, the current continuation can include C parts, as long as they are outside the partial continuation being captured. In other words, the only restriction is that we cannot yield across a C call.

After that realization, and based on Ulrich's proof-of-concept implementation, we were able to implement asymmetrical coroutines in Lua 5.0. The main change was that the interpreter loop, which executes the instructions for the virtual machine, ceased to be recursive. In previous versions, when the interpreter loop executed a CALL instruction, it called itself recursively to execute the called function. Since Lua 5.0, the interpreter behaves more like a real CPU: when it executes a CALL instruction, it pushes some context information onto a call stack and proceeds to execute the called function, restoring the context when that function returns.

After that change, the implementation of coroutines became straightforward.

Unlike most implementations of asymmetrical coroutines, in Lua coroutines are what we call *stackfull* [19]. With them, we can implement symmetrical coroutines and even the call/1cc operator (*call with current one-shot continuation*) proposed for Scheme [11]. However, the use of C functions is severely restricted within these implementations.

We hope that the introduction of coroutines in Lua 5.0 marks a revival of coroutines as powerful control structures [18].

6.8 Extensible semantics

As mentioned in §5.2, we introduced extensible semantics in Lua 2.1 in the form of *fallbacks* as a general mechanism to allow the programmer to intervene whenever Lua did not know how to proceed. Fallbacks thus provided a restricted form of resumable exception handling. In particular, by using fallbacks, we could make a value respond to operations not originally meant for it or make a value of one type behave like a value of another type. For instance, we could make userdata and tables respond to arithmetic operations, userdata behave as tables, strings behave as functions, etc. Moreover, we could make a table respond to keys that were absent in it, which is fundamental for implementing inheritance. With fallbacks for table indexing and a little syntactic sugar for defining and calling methods, object-oriented programming with inheritance became possible in Lua.

Although objects, classes, and inheritance were not core concepts in Lua, they could be implemented directly in Lua, in many flavors, according to the needs of the application. In other words, Lua provided mechanisms, not policy — a tenet that we have tried to follow closely ever since.

The simplest kind of inheritance is inheritance by delegation, which was introduced by Self and adopted in other prototype-based languages such as NewtonScript and JavaScript. The code below shows an implementation of inheritance by delegation in Lua 2.1.

```
function Index(a,i)
  if i == "parent" then
    return nil
  end
  local p = a.parent
  if type(p) == "table" then
    return p[i]
  else
    return nil
  end
end
setfallback("index", Index)
```

When a table was accessed for an absent field (be it an attribute or a method), the index fallback was triggered. Inheritance was implemented by setting the index fallback to follow a chain of "parents" upwards, possibly triggering

the index fallback again, until a table had the required field or the chain ended.

After setting that index fallback, the code below printed 'red' even though 'b' did not have a 'color' field:

```
a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400, parent=a}
print(b.color)
```

There was nothing magical or hard-coded about delegation through a "parent" field. Programmers had complete freedom: they could use a different name for the field containing the parent, they could implement multiple inheritance by trying a list of parents, etc. Our decision not to hard-code any of those possible behaviors led to one of the main design concepts of Lua: *meta-mechanisms*. Instead of littering the language with lots of features, we provided ways for users to program the features themselves, in the way they wanted them, and *only* for those features they needed.

Fallbacks greatly increased the expressiveness of Lua. However, fallbacks were global handlers: there was only one function for each event that could occur. As a consequence, it was difficult to mix different inheritance mechanisms in the same program, because there was only one hook for implementing inheritance (the index fallback). While this might not be a problem for a program written by a single group on top of its own object system, it became a problem when one group tried to use code from other groups, because their visions of the object system might not be consistent with each other. Hooks for different mechanisms could be chained, but chaining was slow, complicated, error-prone, and not very polite. Fallback chaining did not encourage code sharing and reuse; in practice almost nobody did it. This made it very hard to use third-party libraries.

Lua 2.1 allowed userdata to be tagged. In Lua 3.0 we extended tags to all values and replaced fallbacks with *tag methods*. Tag methods were fallbacks that operated only on values with a given tag. This made it possible to implement independent notions of inheritance, for instance. No chaining was needed because tag methods for one tag did not affect tag methods for another tag.

The tag method scheme worked very well and lasted until Lua 5.0, when we replaced tags and tag methods by *metatables* and *metamethods*. Metatables are just ordinary Lua tables and so can be manipulated within Lua without the need for special functions. Like tags, metatables can be used to represent user-defined types with userdata and tables: all objects of the same "type" should share the same metatable. Unlike tags, metatables and their contents are naturally collected when no references remain to them. (In contrast, tags and their tag methods had to live until the end of the program.) The introduction of metatables also simplified the implementation: while tag methods had their own private representation inside Lua's core, metatables use mainly the standard table machinery.

The code below shows the implementation of inheritance in Lua 5.0. The index metamethod replaces the index tag method and is represented by the '__index' field in the metatable. The code makes 'b' inherit from 'a' by setting a metatable for 'b' whose '__index' field points to 'a'. (In general, index metamethods are functions, but we have allowed them to be tables to support simple inheritance by delegation directly.)

```
a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400}
setmetatable(b,{ __index = a })
print(b.color)    --> red
```

6.9 C API

Lua is provided as a library of C functions and macros that allow the host program to communicate with Lua. This API between Lua and C is one of the main components of Lua; it is what makes Lua an embeddable language.

Like the rest of the language, the API has gone through many changes during Lua's evolution. Unlike the rest of the language, however, the API design received little outside influence, mainly because there has been little research activity in this area.

The API has always been bi-directional because, since Lua 1.0, we have considered calling Lua from C and calling C from Lua equally important. Being able to call Lua from C is what makes Lua an *extension language*, that is, a language for extending applications through configuration, macros, and other end-user customizations. Being able to call C from Lua makes Lua an *extensible language*, because we can use C functions to extend Lua with new facilities. (That is why we say that Lua is an extensible extension language [30].) Common to both these aspects are two mismatches between C and Lua to which the API must adjust: static typing in C versus dynamic typing in Lua and manual memory management in C versus automatic garbage collection in Lua.

Currently, the C API solves both difficulties by using an abstract stack[15] to exchange data between Lua and C. Every C function called by Lua gets a new stack frame that initially contains the function arguments. If the C function wants to return values to Lua, it pushes those values onto the stack just before returning.

Each stack slot can hold a Lua value of any type. For each Lua type that has a corresponding representation in C (e.g., strings and numbers), there are two API functions: an *injection* function, which pushes onto the stack a Lua value corresponding to the given C value; and a *projection* function, which returns a C value corresponding to the Lua value at a given stack position. Lua values that have no corresponding representation in C (e.g., tables and functions) can be manipulated via the API by using their stack positions.

---

[15] Throughout this section, 'stack' always means this abstract stack. Lua never accesses the C stack.

Practically all API functions get their operands from the stack and push their results onto the stack. Since the stack can hold values of any Lua type, these API functions operate with any Lua type, thus solving the typing mismatch. To prevent the collection of Lua values in use by C code, the values in the stack are never collected. When a C function returns, its Lua stack frame vanishes, automatically releasing all Lua values that the C function was using. These values will eventually be collected if no further references to them exist. This solves the memory management mismatch.

It took us a long time to arrive at the current API. To discuss how the API evolved, we use as illustration the C equivalent of the following Lua function:

```
function foo(t)
  return t.x
end
```

In words, this function receives a single parameter, which should be a table, and returns the value stored at the 'x' field in that table. Despite its simplicity, this example illustrates three important issues in the API: how to get parameters, how to index tables, and how to return results.

In Lua 1.0, we would write foo in C as follows:

```
void foo_l (void) {
  lua_Object t = lua_getparam(1);
  lua_Object r = lua_getfield(t, "x");
  lua_pushobject(r);
}
```

Note that the required value is stored at the string index "x" because 't.x' is syntactic sugar for 't["x"]'. Note also that all components of the API start with 'lua_' (or 'LUA_') to avoid name clashes with other C libraries.

To export this C function to Lua with the name 'foo' we would do

```
lua_register("foo", foo_l);
```

After that, foo could be called from Lua code just like any other Lua function:

```
t = {x = 200}
print(foo(t))        --> 200
```

A key component of the API was the type lua_Object, defined as follows:

```
typedef struct Object *lua_Object;
```

In words, lua_Object was an abstract type that represented Lua values in C opaquely. Arguments given to C functions were accessed by calling lua_getparam, which returned a lua_Object. In the example, we call lua_getparam once to get the table, which is supposed to be the first argument to foo. (Extra arguments are silently ignored.) Once the table is available in C (as a lua_Object), we get the value of its "x" field by calling lua_getfield. This value is also represented in C as a lua_Object, which is finally sent back to Lua by pushing it onto the stack with lua_pushobject.

The *stack* was another key component of the API. It was used to pass values from C to Lua. There was one push function for each Lua type with a direct representation in C: lua_pushnumber for numbers, lua_pushstring for strings, and lua_pushnil, for the special value nil. There was also lua_pushobject, which allowed C to pass back to Lua an arbitrary Lua value. When a C function returned, all values in the stack were returned to Lua as the results of the C function (functions in Lua can return multiple values).

Conceptually, a lua_Object was a union type, since it could refer to any Lua value. Several scripting languages, including Perl, Python, and Ruby, still use a union type to represent their values in C. The main drawback of this representation is that it is hard to design a garbage collector for the language. Without extra information, the garbage collector cannot know whether a value has a reference to it stored as a union in the C code. Without this knowledge, the collector may collect the value, making the union a dangling pointer. Even when this union is a local variable in a C function, this C function can call Lua again and trigger garbage collection.

Ruby solves this problem by inspecting the C stack, a task that cannot be done in a portable way. Perl and Python solve this problem by providing explicit reference-count functions for these union values. Once you increment the reference count of a value, the garbage collector will not collect that value until you decrement the count to zero. However, it is not easy for the programmer to keep these reference counts right. Not only is it easy to make a mistake, but it is difficult to find the error later (as anyone who has ever debugged memory leaks and dangling pointers can attest). Furthermore, reference counting cannot deal with cyclic data structures that become garbage.

Lua never provided such reference-count functions. Before Lua 2.1, the best you could do to ensure that an unanchored lua_Object was not collected was to avoid calling Lua whenever you had a reference to such a lua_Object. (As long as you could ensure that the value referred to by the union was also stored in a Lua variable, you were safe.) Lua 2.1 brought an important change: it kept track of all lua_Object values passed to C, ensuring that they were not collected while the C function was active. When the C function returned to Lua, then (and only then) all references to these lua_Object values were released, so that they could be collected.[16]

More specifically, in Lua 2.1 a lua_Object ceased to be a pointer to Lua's internal data structures and became an index into an internal array that stored all values that had to be given to C:

```
typedef unsigned int lua_Object;
```

This change made the use of lua_Object reliable: while a value was in that array, it would not be collected by Lua.

---

[16] A similar method is used by JNI to handle "local references".

When the C function returned, its whole array was erased, and the values used by the function could be collected if possible. (This change also gave more freedom for implementing the garbage collector, because it could move objects if necessary; however, we did not followed this path.)

For simple uses, the Lua 2.1 behavior was very practical: it was safe and the C programmer did not have to worry about reference counts. Each `lua_Object` behaved like a local variable in C: the corresponding Lua value was guaranteed to be alive during the lifetime of the C function that produced it. For more complex uses, however, this simple scheme had two shortcomings that demanded extra mechanisms: sometimes a `lua_Object` value had to be locked for longer than the lifetime of the C function that produced it; sometimes it had to be locked for a shorter time.

The first of those shortcomings had a simple solution: Lua 2.1 introduced a system of *references*. The function `lua_lock` got a Lua value from the stack and returned a reference to it. This reference was an integer that could be used any time later to retrieve that value, using the `lua_getlocked` function. (There was also a `lua_unlock` function, which destroyed a reference.) With such references, it was easy to keep Lua values in non-local C variables.

The second shortcoming was more subtle. Objects stored in the internal array were released only when the function returned. If a function used too many values, it could overflow the array or cause an out-of-memory error. For instance, consider the following higher-order iterator function, which repeatedly calls a function and prints the result until the call returns nil:

```
void l_loop (void) {
  lua_Object f = lua_getparam(1);
  for (;;) {
    lua_Object res;
    lua_callfunction(f);
    res = lua_getresult(1);
    if (lua_isnil(res)) break;
    printf("%s\n", lua_getstring(res));
  }
}
```

The problem with this code was that the string returned by each call could not be collected until the end of the loop (that is, of the whole C function), thus opening the possibility of array overflow or memory exhaustion. This kind of error can be very difficult to track, and so the implementation of Lua 2.1 set a hard limit on the size of the internal array that kept `lua_Object` values alive. That made the error easier to track because Lua could say "too many objects in a C function" instead of a generic out-of-memory error, but it did not avoid the problem.

To address the problem, the API in Lua 2.1 offered two functions, `lua_beginblock` and `lua_endblock`, that created dynamic scopes ("blocks") for `lua_Object` values;

all values created after a `lua_beginblock` were removed from the internal array at the corresponding `lua_endblock`. However, since a block discipline could not be forced onto C programmers, it was all too common to forget to use these blocks. Moreover, such explicit scope control was a little tricky to use. For instance, a naive attempt to correct our previous example by enclosing the `for` body within a block would fail: we had to call `lua_endblock` just before the `break`, too. This difficulty with the scope of Lua objects persisted through several versions and was solved only in Lua 4.0, when we redesigned the whole API. Nevertheless, as we said before, for typical uses the API was very easy to use, and most programmers never faced the kind of situation described here. More important, the API was safe. Erroneous use could produce well-defined errors, but not dangling references or memory leaks.

Lua 2.1 brought other changes to the API. One was the introduction of `lua_getsubscript`, which allowed the use of any value to index a table. This function had no explicit arguments: it got both the table and the key from the stack. The old `lua_getfield` was redefined as a macro, for compatibility:

```
#define lua_getfield(o,f)  \
   (lua_pushobject(o), lua_pushstring(f), \
    lua_getsubscript())
```

(Backward compatibility of the C API is usually implemented using macros, whenever feasible.)

Despite all those changes, syntactically the API changed little from Lua 1 to Lua 2. For instance, our illustrative function `foo` could be written in Lua 2 exactly as we wrote it for Lua 1.0. The meaning of `lua_Object` was quite different, and `lua_getfield` was implemented on top of new primitive operations, but for the average user it was as if nothing had changed. Thereafter, the API remained fairly stable until Lua 4.0.

Lua 2.4 expanded the reference mechanism to support weak references. A common design in Lua programs is to have a Lua object (typically a table) acting as a proxy for a C object. Frequently the C object must know who its proxy is and so keeps a reference to the proxy. However, that reference prevents the collection of the proxy object, even when the object becomes inaccessible from Lua. In Lua 2.4, the program could create a *weak reference* to the proxy; that reference did not prevent the collection of the proxy object. Any attempt to retrieve a collected reference resulted in a special value `LUA_NOOBJECT`.

Lua 4.0 brought two main novelties in the C API: support for multiple Lua states and a virtual stack for exchanging values between C and Lua. Support for multiple, independent Lua states was achieved by eliminating all global state. Until Lua 3.0, only one Lua state existed and it was implemented using many static variables scattered throughout the code. Lua 3.1 introduced multiple independent Lua states; all static variables were collected into a single C struct. An

API function was added to allow switching states, but only one Lua state could be active at any moment. All other API functions operated over the active Lua state, which remained implicit and did not appear in the calls. Lua 4.0 introduced explicit Lua states in the API. This created a big incompatibility with previous versions.[17] All C code that communicated with Lua (in particular, all C functions registered to Lua) had to be changed to include an explicit state argument in calls to the C API. Since all C functions had to be rewritten anyway, we took this opportunity and made another major change in the C–Lua communication in Lua 4.0: we replaced the concept of `lua_Object` by an explicit virtual stack used for all communication between Lua and C in both directions. The stack could also be used to store temporary values.

In Lua 4.0, our `foo` example could be written as follows:

```
int foo_l (lua_State *L) {
  lua_pushstring(L, "x");
  lua_gettable(L, 1);
  return 1;
}
```

The first difference is the function signature: `foo_l` now receives a Lua state on which to operate and returns the number of values returned by the function in the stack. In previous versions, all values left in the stack when the function ended were returned to Lua. Now, because the stack is used for all operations, it can contain intermediate values that are not to be returned, and so the function needs to tell Lua how many values in the stack to consider as return values. Another difference is that `lua_getparam` is no longer needed, because function arguments come in the stack when the function starts and can be directly accessed by their index, like any other stack value.

The last difference is the use of `lua_gettable`, which replaced `lua_getsubscript` as the means to access table fields. `lua_gettable` receives the table to be indexed as a stack position (instead of as a Lua object), pops the key from the top of the stack, and pushes the result. Moreover, it leaves the table in the same stack position, because tables are frequently indexed repeatedly. In `foo_l`, the table used by `lua_gettable` is at stack position 1, because it is the first argument to that function, and the key is the string `"x"`, which needs to be pushed onto the stack before calling `lua_gettable`. That call replaces the key in the stack with the corresponding table value. So, after `lua_gettable`, there are two values in the stack: the table at position 1 and the result of the indexing at position 2, which is the top of the stack. The C function returns 1 to tell Lua to use that top value as the single result returned by the function.

To further illustrate the new API, here is an implementation of our loop example in Lua 4.0:

```
int l_loop (lua_State *L) {
  for (;;) {
    lua_pushvalue(L, 1);
    lua_call(L, 0, 1);
    if (lua_isnil(L, -1)) break;
    printf("%s\n", lua_tostring(L, -1));
    lua_pop(L, 1);
  }
  return 0;
}
```

To call a Lua function, we push it onto the stack and then push its arguments, if any (none in the example). Then we call `lua_call`, telling how many arguments to get from the stack (and therefore implicitly also telling where the function is in the stack) and how many results we want from the call. In the example, we have no arguments and expect one result. The `lua_call` function removes the function and its arguments from the stack and pushes back exactly the requested number of results. The call to `lua_pop` removes the single result from the stack, leaving the stack at the same level as at the beginning of the loop. For convenience, we can index the stack from the bottom, with positive indices, or from the top, with negative indices. In the example, we use index -1 in `lua_isnil` and `lua_tostring` to refer to the top of the stack, which contains the function result.

With hindsight, the use of a single stack in the API seems an obvious simplification, but when Lua 4.0 was released many users complained about the complexity of the new API. Although Lua 4.0 had a much cleaner conceptual model for its API, the direct manipulation of the stack requires some thought to get right. Many users were content to use the previous API without any clear conceptual model of what was going on behind the scenes. Simple tasks did not require a conceptual model at all and the previous API worked quite well for them. More complex tasks often broke whatever private models users had, but most users never programmed complex tasks in C. So, the new API was seen as too complex at first. However, such skepticism gradually vanished, as users came to understand and value the new model, which proved to be simpler and much less error-prone.

The possibility of multiple states in Lua 4.0 created an unexpected problem for the reference mechanism. Previously, a C library that needed to keep some object fixed could create a reference to the object and store that reference in a global C variable. In Lua 4.0, if a C library was to work with several states, it had to keep an individual reference for each state and so could not keep the reference in a global C variable. To solve this difficulty, Lua 4.0 introduced the *registry*, which is simply a regular Lua table available to C only. With the registry, a C library that wants to keep a Lua object can choose a unique key and associate the object with this key in the registry. Because each independent Lua state has its own registry, the C library can use the same key in each state to manipulate the corresponding object.

---

[17] We provided a module that emulated the 3.2 API on top of the 4.0 API, but we do not think it was used much.

We could quite easily implement the original reference mechanism on top of the registry by using integer keys to represent references. To create a new reference, we just find an unused integer key and store the value at that key. Retrieving a reference becomes a simple table access. However, we could not implement weak references using the registry. So, Lua 4.0 kept the previous reference mechanism. In Lua 5.0, with the introduction of weak tables in the language, we were finally able to eliminate the reference mechanism from the core and move it to a library.

The C API has slowly evolved toward completeness. Since Lua 4.0, all standard library functions can be written using only the C API. Until then, Lua had a number of built-in functions (from 7 in Lua 1.1 to 35 in Lua 3.2), most of which could have been written using the C API but were not because of a perceived need for speed. A few built-in functions could not have been written using the C API because the C API was not complete. For instance, until Lua 3.2 it was not possible to iterate over the contents of a table using the C API, although it was possible to do it in Lua using the built-in function `next`. The C API is not yet complete and not everything that can be done in Lua can be done in C; for instance, the C API lacks functions for performing arithmetic operations on Lua values. We plan to address this issue in the next version.

6.10   Userdata

Since its first version, an important feature of Lua has been its ability to manipulate C data, which is provided by a special Lua data type called *userdata*. This ability is an essential component in the extensibility of Lua.

For Lua programs, the userdata type has undergone no changes at all throughout Lua's evolution: although userdata are first-class values, userdata is an opaque type and its only valid operation in Lua is equality test. Any other operation over userdata (creation, inspection, modification) must be provided by C functions.

For C functions, the userdata type has undergone several changes in Lua's evolution. In Lua 1.0, a userdata value was a simple `void*` pointer. The main drawback of this simplicity was that a C library had no way to check whether a userdata was valid. Although Lua code cannot create userdata values, it can pass userdata created by one library to another library that expects pointers to a different structure. Because C functions had no mechanisms to check this mismatch, the result of this pointer mismatch was usually fatal to the application. We have always considered it unacceptable for a Lua program to be able to crash the host application. Lua should be a safe language.

To overcome the pointer mismatch problem, Lua 2.1 introduced the concept of *tags* (which would become the seed for *tag methods* in Lua 3.0). A tag was simply an arbitrary integer value associated with a userdata. A userdata's tag could only be set once, when the userdata was created. Provided that each C library used its own exclusive tag, C code could

easily ensure that a userdata had the expected type by checking its tag. (The problem of how a library writer chose a tag that did not clash with tags from other libraries remained open. It was only solved in Lua 3.0, which provided tag management via `lua_newtag`.)

A bigger problem with Lua 2.1 was the management of C resources. More often than not, a userdata pointed to a dynamically allocated structure in C, which had to be freed when its corresponding userdata was collected in Lua. However, userdata were values, not objects. As such, they were not collected (in the same way that numbers are not collected). To overcome this restriction, a typical design was to use a table as a proxy for the C structure in Lua, storing the actual userdata in a predefined field of the proxy table. When the table was collected, its finalizer would free the corresponding C structure.

This simple solution created a subtle problem. Because the userdata was stored in a regular field of the proxy table, a malicious user could tamper with it from within Lua. Specifically, a user could make a copy of the userdata and use the copy after the table was collected. By that time, the corresponding C structure had been destroyed, making the userdata a dangling pointer, with disastrous results. To improve the control of the life cycle of userdata, Lua 3.0 changed userdata from values to objects, subject to garbage collection. Users could use the userdata finalizer (the garbage-collection tag method) to free the corresponding C structure. The correctness of Lua's garbage collector ensured that a userdata could not be used after being collected.

However, userdata as objects created an identity problem. Given a userdata, it is trivial to get its corresponding pointer, but frequently we need to do the reverse: given a C pointer, we need to get its corresponding userdata.[18] In Lua 2, two userdata with the same pointer and the same tag would be equal; equality was based on their values. So, given the pointer and the tag, we had the userdata. In Lua 3, with userdata being objects, equality was based on identity: two userdata were equal only when they were the *same* userdata (that is, the same object). Each userdata created was different from all others. Therefore, a pointer and a tag would not be enough to get the corresponding userdata.

To solve this difficulty, and also to reduce incompatibilities with Lua 2, Lua 3 adopted the following semantics for the operation of pushing a userdata onto the stack: if Lua already had a userdata with the given pointer and tag, then that userdata was pushed on the stack; otherwise, a new userdata was created and pushed on the stack. So, it was easy for C code to translate a C pointer to its corresponding userdata in Lua. (Actually, the C code could be the same as it was in Lua 2.)

---

[18] A typical scenario for this need is the handling of callbacks in a GUI toolkit. The C callback associated with a widget gets only a pointer to the widget, but to pass this callback to Lua we need the userdata that represents that widget in Lua.

However, Lua 3 behavior had a major drawback: it combined into a single primitive (`lua_pushuserdata`) two basic operations: userdata searching and userdata creation. For instance, it was impossible to check whether a given C pointer had a corresponding userdata without creating that userdata. Also, it was impossible to create a new userdata regardless of its C pointer. If Lua already had a userdata with that value, no new userdata would be created.

Lua 4 mitigated that drawback by introducing a new function, `lua_newuserdata`. Unlike `lua_pushuserdata`, this function always created a new userdata. Moreover, what was more important at that time, those userdata were able to store arbitrary C data, instead of pointers only. The user would tell `lua_newuserdata` the amount memory to be allocated and `lua_newuserdata` returned a pointer to the allocated area. By having Lua allocate memory for the user, several common tasks related to userdata were simplified. For instance, C code did not need to handle memory-allocation errors, because they were handled by Lua. More important, C code did not need to handle memory deallocation: memory used by such userdata was released by Lua automatically, when the userdata was collected.

However, Lua 4 still did not offer a nice solution to the search problem (i.e., finding a userdata given its C pointer). So, it kept the `lua_pushuserdata` operation with its old behavior, resulting in a hybrid system. It was only in Lua 5 that we removed `lua_pushuserdata` and dissociated userdata creation and searching. Actually, Lua 5 removed the searching facility altogether. Lua 5 also introduced *light userdata*, which store plain C pointer values, exactly like regular userdata in Lua 1. A program can use a weak table to associate C pointers (represented as light userdata) to its corresponding "heavy" userdata in Lua.

As is usual in the evolution of Lua, userdata in Lua 5 is more flexible than it was in Lua 4; it is also simpler to explain and simpler to implement. For simple uses, which only require storing a C structure, userdata in Lua 5 is trivial to use. For more complex needs, such as those that require mapping a C pointer back to a Lua userdata, Lua 5 offers the mechanisms (light userdata and weak tables) for users to implement strategies suited to their applications.

6.11   Reflectivity

Since its very first version Lua has supported some reflective facilities. A major reason for this support was the proposed use of Lua as a configuration language to replace SOL. As described in §4, our idea was that the programmer could use the language itself to write type-checking routines, if needed.

For instance, if a user wrote something like

```
T = @track{ y=9, x=10, id="1992-34" }
```

we wanted to be able to check that the track did have a y field and that this field was a number. We also wanted to be able to check that the track did not have extraneous fields

(possibly to catch typing mistakes). For these two tasks, we needed access to the type of a Lua value and a mechanism to traverse a table and visit all its pairs.

Lua 1.0 provided the needed functionality with only two functions, which still exist: `type` and `next`. The `type` function returns a string describing the type of any given value ("number", "nil", "table", etc.). The `next` function receives a table and a key and returns a "next" key in the table (in an arbitrary order). The call `next(t,nil)` returns a "first" key. With `next` we can traverse a table and process all its pairs. For instance, the following code prints all pairs in a table `t`:[19]

```
k = next(t,nil)
while k do
  print(k,t[k])
  k = next(t,k)
end
```

Both these functions have a simple implementation: `type` checks the internal tag of the given value and returns the corresponding string; `next` finds the given key in the table and then goes to the next key, following the internal table representation.

In languages like Java and Smalltalk, reflection must reify concepts like classes, methods, and instance variables. Moreover, that reification demands new concepts like metaclasses (the class of a reified class). Lua needs nothing like that. In Lua, most facilities provided by the Java reflective package come for free: classes and modules are tables, methods are functions. So, Lua does not need any special mechanism to reify them; they are plain program values. Similarly, Lua does not need special mechanisms to build method calls at run time (because functions are first-class values and Lua's parameter-passing mechanism naturally supports calling a function with a variable number of arguments), and it does not need special mechanisms to access a global variable or an instance variable given its name (because they are regular table fields).[20]

7.   Retrospect

In this section we give a brief critique of Lua's evolutionary process, discussing what has worked well, what we regret, and what we do not really regret but could have done differently.

One thing that has worked really well was the early decision (made in Lua 1.0) to have tables as the sole data-structuring mechanism in Lua. Tables have proved to be powerful and efficient. The central role of tables in the language and in its implementation is one of the main character-

---

[19] Although this code still works, the current idiom is 'for k,v in pairs(t) do print(k,v) end'.

[20] Before Lua 4.0, global variables were stored in a special data structure inside the core, and we provided a `nextvar` function to traverse it. Since Lua 4.0, global variables are stored in a regular Lua table and `nextvar` is no longer needed.

istics of Lua. We have resisted user pressure to include other data structures, mainly "real" arrays and tuples, first by being stubborn, but also by providing tables with an efficient implementation and a flexible design. For instance, we can represent a set in Lua by storing its elements as indices of a table. This is possible only because Lua tables accept any value as index.

Another thing that has worked well was our insistence on portability, which was initially motivated by the diverse platforms of Tecgraf's clients. This allowed Lua to be compiled for platforms we had never dreamed of supporting. In particular, Lua's portability is one of the reasons that Lua has been widely adopted for developing games. Restricted environments, such as game consoles, tend not to support the complete semantics of the full standard C library. By gradually reducing the dependency of Lua's core on the standard C library, we are moving towards a Lua core that requires only a free-standing ANSI C implementation. This move aims mainly at embedding flexibility, but it also increases portability. For instance, since Lua 3.1 it is easy to change a few macros in the code to make Lua use an application-specific memory allocator, instead of relying on `malloc` and friends. Starting with Lua 5.1, the memory allocator can be provided dynamically when creating a Lua state.

With hindsight, we consider that being raised by a small committee has been very positive for the evolution of Lua. Languages designed by large committees tend to be too complicated and never quite fulfill the expectations of their sponsors. Most successful languages are raised rather than designed. They follow a slow bottom-up process, starting as a small language with modest goals. The language evolves as a consequence of actual feedback from real users, from which design flaws surface and new features that are actually useful are identified. This describes the evolution of Lua quite well. We listen to users and their suggestions, but we include a new feature in Lua only when all three of us agree; otherwise, it is left for the future. It is much easier to add features later than to remove them. This development process has been essential to keep the language simple, and simplicity is our most important asset. Most other qualities of Lua — speed, small size, and portability — derive from its simplicity.

Since its first version Lua has had real users, that is, users others than ourselves, who care not about the language itself but only about how to use it productively. Users have always given important contributions to the language, through suggestions, complaints, use reports, and questions. Again, our small committee plays an important role in managing this feedback: its structure gives us enough inertia to listen closely to users without having to follow all their suggestions.

Lua is best described as a closed-development, open-source project. This means that, even though the source code is freely available for scrutiny and adaption, Lua is not developed in a collaborative way. We do accept user suggestions, but never their code verbatim. We always try to do our own implementation.

Another unusual aspect of Lua's evolution has been our handling of incompatible changes. For a long time we considered simplicity and elegance more important than compatibility with previous versions. Whenever an old feature was superseded by a new one, we simply removed the old feature. Frequently (but not always), we provided some sort of compatibility aid, such as a compatibility library, a conversion script, or (more recently) compile-time options to preserve the old feature. In any case, the user had to take some measures when moving to a new version.

Some upgrades were a little traumatic. For instance, Tecgraf, Lua's birthplace, never upgraded from Lua 3.2 to Lua 4.0 because of the big changes in the API. Currently, a few Tecgraf programs have been updated to Lua 5.0, and new programs are written in this version, too. But Tecgraf still has a large body of code in Lua 3.2. The small size and simplicity of Lua alleviates this problem: it is easy for a project to keep to an old version of Lua, because the project group can do its own maintenance of the code, when necessary.

We do not really regret this evolution style. Gradually, however, we have become more conservative. Not only is our user and code base much larger than it once was, but also we feel that Lua as a language is much more mature.

We should have introduced booleans from the start, but we wanted to start with the simplest possible language. Not introducing booleans from the start had a few unfortunate side-effects. One is that we now have two false values: nil and false. Another is that a common protocol used by Lua functions to signal errors to their callers is to return nil followed by an error message. It would have been better if false had been used instead of nil in that case, with nil being reserved for its primary role of signaling the absence of any useful value.

Automatic coercion of strings to numbers in arithmetic operations, which we took from Awk, could have been omitted. (Coercion of numbers to strings in string operations is convenient and less troublesome.)

Despite our "mechanisms, not policy" rule — which we have found valuable in guiding the evolution of Lua — we should have provided a precise set of policies for modules and packages earlier. The lack of a common policy for building modules and installing packages prevents different groups from sharing code and discourages the development of a community code base. Lua 5.1 provides a set of policies for modules and packages that we hope will remedy this situation.

As mentioned in §6.4, Lua 3.0 introduced support for conditional compilation, mainly motivated to provide a means to disable code. We received many requests for enhancing conditional compilation in Lua, even by people who did not

use it! By far the most popular request was for a full macro processor like the C preprocessor. Providing such a macro processor in Lua would be consistent with our general philosophy of providing extensible mechanisms. However, we would like it to be programmable in Lua, not in some other specialized language. We did not want to add a macro facility directly into the lexer, to avoid bloating it and slowing compilation. Moreover, at that time the Lua parser was not fully reentrant, and so there was no way to call Lua from within the lexer. (This restriction was removed in Lua 5.1.) So endless discussions ensued in the mailing list and within the Lua team. But no consensus was ever reached and no solution emerged. We still have not completely dismissed the idea of providing Lua with a macro system: it would give Lua extensible syntax to go with extensible semantics.

8. Conclusion

Lua has been used successfully in many large companies, such as Adobe, Bombardier, Disney, Electronic Arts, Intel, LucasArts, Microsoft, Nasa, Olivetti, and Philips. Many of these companies have shipped Lua embedded into commercial products, often exposing Lua scripts to end users.

Lua has been especially successful in games. It was said recently that "Lua is rapidly becoming the de facto standard for game scripting" [37]. Two informal polls [5, 6] conducted by gamedev.net (an important site for game programmers) in September 2003 and in June 2006 showed Lua as the most popular scripting language for game development. Roundtables dedicated to Lua in game development were held at GDC in 2004 and 2006. Many famous games use Lua: Baldur's Gate, Escape from Monkey Island, FarCry, Grim Fandango, Homeworld 2, Illarion, Impossible Creatures, Psychonauts, The Sims, World of Warcraft. There are two books on game development with Lua [42, 25], and several other books on game development devote chapters to Lua [23, 44, 41, 24].

The wide adoption of Lua in games came as a surprise to us. We did not have game development as a target for Lua. (Tecgraf is mostly concerned with scientific software.) With hindsight, however, that success is understandable because all the features that make Lua special are important in game development:

Portability: Many games run on non-conventional platforms, such as game consoles, that need special development tools. An ANSI C compiler is all that is needed to build Lua.

Ease of embedding: Games are demanding applications. They need both performance, for its graphics and simulations, and flexibility, for the creative staff. Not by chance, many games are coded in (at least) two languages, one for scripting and the other for coding the engine. Within that framework, the ease of integrating Lua with another language (mainly C++, in the case of games) is a big advantage.

Simplicity: Most game designers, scripters and level writers are not professional programmers. For them, a language with simple syntax and simple semantics is particularly important.

Efficiency and small size: Games are demanding applications; the time alloted to running scripts is usually quite small. Lua is one of the fastest scripting languages [1]. Game consoles are restricted environments. The script interpreter should be parsimonious with resources. The Lua core takes about 100K.

Control over code: Unlike most other software enterprises, game production involves little evolution. In many cases, once a game has been released, there are no updates or new versions, only new games. So, it is easier to risk using a new scripting language in a game. Whether the scripting language will evolve or how it will evolve is not a crucial point for game developers. All they need is the version they used in the game. Since they have complete access to the source code of Lua, they can simply keep the same Lua version forever, if they so choose.

Liberal license: Most commercial games are not open source. Some game companies even refuse to use any kind of open-source code. The competition is hard, and game companies tend to be secretive about their technologies. For them, a liberal license like the Lua license is quite convenient.

Coroutines: It is easier to script games if the scripting language supports multitasking because a character or activity can be suspended and resumed later. Lua supports cooperative multitasking in the form of coroutines [14].

Procedural data files: Lua's original design goal of providing powerful data-description facilities allows games to use Lua for data files, replacing special-format textual data files with many benefits, especially homogeneity and expressiveness.

Acknowledgments

References

[1] The computer language shootout benchmarks. `http://shootout.alioth.debian.org/`.

[2] Lua projects. `http://www.lua.org/uses.html`.

[3] The MIT license. `http://www.opensource.org/licenses/mit-license.html`.

[4] Timeline of programming languages. `http://en.wikipedia.org/wiki/Timeline_of_programming_languages`.

[5] Which language do you use for scripting in your game engine? `http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163`, Sept. 2003.

[6] Which is your favorite embeddable scripting language? `http://www.gamedev.net/gdpolls/viewpoll.asp?ID=788`, June 2006.

[7] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[8] G. Bell, R. Carey, and C. Marrin. The Virtual Reality Modeling Language Specification—Version 2.0. `http://www.vrml.org/VRML2.0/FINAL/`, Aug. 1996. (ISO/IEC CD 14772).

[9] J. Bentley. Programming pearls: associative arrays. *Communications of the ACM*, 28(6):570–576, 1985.

[10] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[11] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–107, 1996.

[12] W. Celes, L. H. de Figueiredo, and M. Gattass. EDG: uma ferramenta para criação de interfaces gráficas interativas. In *Proceedings of SIBGRAPI '95 (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 241–248, 1995.

[13] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49. ACM Press, 2003.

[14] L. H. de Figueiredo, W. Celes, and R. Ierusalimschy. Programming advanced control mechanisms with Lua coroutines. In *Game Programming Gems 6*, pages 357–369. Charles River Media, 2006.

[15] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. The design and implementation of a language for extending applications. In *Proceedings of XXI SEMISH (Brazilian Seminar on Software and Hardware)*, pages 273–284, 1994.

[16] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, Dec. 1996.

[17] L. H. de Figueiredo, C. S. Souza, M. Gattass, and L. C. G. Coelho. Geração de interfaces para captura de dados sobre desenhos. In *Proceedings of SIBGRAPI '92 (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 169–175, 1992.

[18] A. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[19] A. L. de Moura and R. Ierusalimschy. Revisiting coroutines. MCC 15/04, PUC-Rio, 2004.

[20] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1987. Technical Report #87-011.

[21] M. Feeley and G. Lapalme. Closure generation based on viewing LAMBDA as EPSILON plus COMPILE. *Journal of Computer Languages*, 17(4):251–267, 1992.

[22] T. G. Gorham and R. Ierusalimschy. Um sistema de depuração reflexivo para uma linguagem de extensão. In *Anais do I Simpósio Brasileiro de Linguagens de Programação*, pages 103–114, 1996.

[23] T. Gutschmidt. *Game Programming with Python, Lua, and Ruby*. Premier Press, 2003.

[24] M. Harmon. Building Lua into games. In *Game Programming Gems 5*, pages 115–128. Charles River Media, 2005.

[25] J. Heiss. *Lua Scripting für Spieleprogrammierer. Hit the Ground with Lua*. Stefan Zerbst, Dec. 2005.

[26] A. Hester, R. Borges, and R. Ierusalimschy. Building flexible and extensible web applications with Lua. *Journal of Universal Computer Science*, 4(9):748–762, 1998.

[27] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2003.

[28] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2nd edition, 2006.

[29] R. Ierusalimschy, W. Celes, L. H. de Figueiredo, and R. de Souza. Lua: uma linguagem para customização de aplicações. In *VII Simpósio Brasileiro de Engenharia de Software — Caderno de Ferramentas*, page 55, 1993.

[30] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua: an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

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

[32] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. *Lua 5.1 Reference Manual*. Lua.org, 2006.

[33] K. Jung and A. Brown. *Beginning Lua Programming*. Wrox, 2007.

[34] L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1986.

[35] M. J. Lima and R. Ierusalimschy. Continuações em Lua. In *VI Simpósio Brasileiro de Linguagens de Programação*, pages 218–232, June 2002.

[36] D. McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped LISP. In *ACM conference on LISP and functional programming*, pages 154–162, 1980.

[37] I. Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, 2006.

[38] B. Mogilefsky. Lua in Grim Fandango. `http://www.grimfandango.net/?page=articles&pagenumber=2`, May 1999.

[39] Open Software Foundation. *OSF/Motif Programmer's Guide*. Prentice-Hall, Inc., 1991.

[40] J. Ousterhout. Tcl: an embeddable command language. In *Proc. of the Winter 1990 USENIX Technical Conference*. USENIX Association, 1990.

[41] D. Sanchez-Crespo. *Core Techniques and Algorithms in Game Programming*. New Riders Games, 2003.

[42] P. Schuytema and M. Manyen. *Game Development with Lua*. Delmar Thomson Learning, 2005.

[43] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[44] A. Varanese. *Game Scripting Mastery*. Premier Press, 2002.