Programming with Multiple Paradigms in Lua

Roberto Ierusalimschy

PUC-Rio, Rio de Janeiro, Brazil
roberto@inf.puc-rio.br

Abstract. Lua is a scripting language used in many industrial applications, with an emphasis on embedded systems and games. Two key points in the design of the language that led to its widely adoption are flexibility and small size. To achieve these two conflicting goals, the design emphasizes the use of few but powerful mechanisms, such as first-class functions, associative arrays, coroutines, and reflexive capabilities. As a consequence of this design, although Lua is primarily a procedural language, it is frequently used in several different programming paradigms, such as functional, object-oriented, goal-oriented, and concurrent programming, and also for data description.
In this paper we discuss what mechanisms Lua features to achieve its flexibility and how programmers use them for different paradigms.

1 Introduction

Lua is an embeddable scripting language used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems and games. It is embedded in devices ranging from cameras (Canon) to keyboards (Logitech G15) to network security appliances (Cisco ASA). In 2003 it was voted the most popular language for scripting games by a poll on the site Gamedev[1]. In 2006 it was called a "de facto standard for game scripting" [1]. Lua is also part of the Brazilian standard middleware for digital TV [2].

Like many other languages, Lua strives to be a flexible language. However, Lua also strives to be a small language, both in terms of its specification and its implementation. This is an important feature for an embeddable language that frequently is used in devices with limited hardware resources [3]. To achieve these two conflicting goals, the design of Lua has always been economical about new features. It emphasizes the use of few but powerful mechanisms, such as first-class functions, associative arrays, coroutines, and reflexive capabilities [4, 5].

Lua has several similarities with Scheme, despite a very different syntax. (Lua adopts a conventional syntax instead of Lisp's S-expressions.) Both languages are dynamically typed. As in Scheme, all functions in Lua are anonymous first-class values with lexical scoping; a "function name" is just the name of a regular variable that refers to that function. As in Scheme, Lua does proper tail calls. Lua also offers a single unifying data-structure mechanism.

[1] http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163

However, to keep the language and its implementation small, Lua is more pragmatic than Scheme. Its main data structure is the *table*, or associative arrays, instead of lists. (The former seems a better fit for procedural programming, while the latter seems better for functional programming.) Instead of a hierarchy of numerical types —real, rational, integer— in Lua all numbers are double floating-point values. (With this representation, Lua transfers all the burden of numeric specification and implementation to the underlying system.) Instead of full continuations, Lua offers one-shot continuations in the form of *stackfull* coroutines [6]. (Efficient implementations of coroutines are much simpler than efficient implementations of full continuations and most uses of continuations can be done with coroutines/one-shot continuations.)

As an authentic scripting language, a design goal of Lua is to offer strong support for dual-language programming [7]. The API with C is a key ingredient of the language. To easy the integration between the scripting language and the host language, Lua is amenable to different kinds of programming: event-driven, object oriented, etc. Moreover, to better explore the flexibility offered by the language, Lua programmers frequently use several paradigms, such as functional, object-oriented, goal-oriented, and concurrent programming, and also data description.

In this paper we discuss what mechanisms Lua features to achieve its flexibility and how programmers use them for different paradigms. The rest of the paper is organized around different paradigms. The next section describes the uses of Lua for data description. Section 3 discusses Lua support for functional programming. Section 4 discusses object-oriented programming in Lua. Section 5 shows how we can use coroutines to implement goal-oriented programming, where a *goal* is either a primitive goal or a disjunction of alternative goals. Section 6 discusses two ways to implement concurrency in Lua: *collaborative multithreading*, using coroutines, and *Lua processes*, using multiple independent states. Finally, Section 7 draws some conclusions.

2 Data Description

Lua was born from a data-description language, called SOL [8], a language somewhat similar to XML in intent. Lua inherited from SOL the support for data description, but integrated that support into its procedural semantics.

SOL was somewhat inspired by BibTeX, a tool for creating and formating lists of bibliographic references [9]. A main difference between SOL and BibTeX was that SOL had the ability to name and nest declarations. Figure 1 shows a typical fragment of SOL code, slightly adapted to meet the current syntax of Lua. SOL acted like an XML DOM reader, reading the data file and building an internal tree representing that data; an application then could use an API to traverse that tree.

Lua mostly kept the original SOL syntax, with small changes. The semantics, however, was very different. In Lua, the code in Figure 1 is an imperative program. The syntax `{first = "Daniel", ...}` is a *constructor*: it builds a table,

```
dan = name{first = "Daniel", last = "Friedman"}

mitch = name{last = "Wand",
             first = "Mitchell",
             middle = "P."}

chris = name{first = "Christopher", last = "Haynes"}

book{"essentials",
  author = {dan, mitch, chris},
  title = "Essentials of Programming Languages",
  edition = 2,
  year = 2001,
  publisher = "The MIT Press"
}
```

Fig. 1. data description with SOL/Lua

or associative array, with the given keys and values. The syntax `name{...}` is sugar for `name({...})`, that is, it builds a table and calls function `name` with that table as the sole argument. The syntax `{dan,mitch,chris}` again builds a table, but this time with implicit integer keys 1, 2, and 3, therefore representing a list. A program loading such a file should previously define functions `name` and `book` with appropriate behavior. For instance, function `book` could add the table to some internal list for later treatment.

Several applications use Lua for data description. Games frequently use Lua to describe characters and scenes. HiQLab, a tool for simulating high frequency resonators, uses Lua to describe finite-element meshes [10]. GUPPY uses Lua to describe sequence annotation data from genome databases [11]. Some descriptions comprise thousands of elements running for a few million lines of code. The user sees these files as data files, but Lua sees them as regular code. These huge "programs" pose a heavy load on the Lua precompiler. To handle such files efficiently, and also for simplicity, Lua uses a one-pass compiler with no intermediate representations. As we will see in the next section, this requirement puts a burden on other aspects of the implementation.

3 Functional Programming

Lua offers first-class functions with lexical scoping. For instance, the following code is valid Lua code:

```
(function (a,b) print(a+b) end)(10, 20)
```

It creates an anonymous function that prints the sum of its two parameters and applies that function to arguments 10 and 20.

All functions in Lua are anonymous dynamic values, created at run time. Lua offers a quite conventional syntax for creating functions, like in the following

definition of a factorial function:

```
function fact (n)
  if n <= 1 then return 1
  else return n * fact(n - 1)
  end
end
```

However, this syntax is simply sugar for an assignment:

```
fact = function (n)
         ...
       end
```

This is quite similar to a `define` in Scheme [12].

Lua does not offer a *letrec* primitive. Instead, it relies on assignment to close a recursive reference. For instance, a strict recursive fixed-point operator can be defined like this:

```
local Y
Y = function (f)
      return function (x)
               return f(Y(f))(x)
             end
    end
```

Or, using some syntactic sugar, like this:

```
local function Y (f)
      return function (x)
               return f(Y(f))(x)
             end
    end
```

This second fragment expands to the first one. In both cases, the `Y` in the function body is bounded to the previously declared local variable.

Of course, we can also define a strict non-recursive fixed-point combinator in Lua:

```
Y = function (le)
      local a = function (f)
        return le(function (x) return f(f)(x) end)
      end
      return a(a)
    end
```

Despite being a procedural language, Lua frequently uses function values. Several functions in the standard Lua library are higher-order. For instance,

the `sort` function accepts a comparison function as argument. In its pattern-matching functions, text substitution accepts a *replacement* function that receives the original text matching the pattern and returns its replacement. The standard library also offers some *traversal functions*, which receive a function to be applied to every element of a collection.

Most programming techniques for strict functional programming also work without modifications in Lua. As an example, LuaSocket, the standard library for network connection in Lua, uses functions to allow easy composition of different functionalities when reading from and writing to sockets [13].

Lua also features proper tail calls. Again, although this is a feature from the functional world, it has several interesting uses in procedural programs. For instance, it is used in a standard technique for implementing state machines [4]. In these implementations, each state is represented by a function, and tail calls transfer the program from one state to another.

Closures

The standard technique for implementing strict first-class functions with lexical scoping is with the use of closures. Most implementations of closures neglect assignment. Pure functional languages do not have assignment. In ML assignable cells have no names, so the problem of assignment to lexical-scoped variables does not arise. Since Rabbit [14], most Scheme compilers do *assignment conversions* [15], that is, they implement assignable variables as ML cells on the correct ground that they are not used often.

None of those implementations fit Lua, a procedural language where assignment is the norm. Moreover, as we already mentioned, Lua has an added requirement that its compiler must be fast, to handle huge data-description "programs", and small. So, Lua uses a simple one-pass compiler with no intermediate representations which cannot perform even escape analysis.

Due to these technical restrictions, previous versions of Lua offered a restricted form of lexical scoping. In that restricted form, a nested function could access the value of an outer variable, but could not assign to such variable. Moreover, the value accessed was frozen when the closure was created. Lua version 5, released in 2003, came with a novel technique for implementing closures that satisfies the following requirements [16]:

- It does not impact the performance of code that does not use non-local variables.
- It has an acceptable performance for imperative programs, where side effects (assignment) are the norm.
- It correctly handles *sharing*, where more than one closure modifies a non-local variable.
- It is compatible with the standard execution model for procedural languages, where variables live in activation records allocated in an array-based stack.
- It is amenable to a one-pass compiler that generates code on the fly, without intermediate representations.

4 Object-Oriented Programming

Lua has only one data-structure mechanism, the *table*. Tables are first-class, dynamically created associative arrays.

Tables plus first-class functions already give Lua partial support for objects. An object may be represented by a table: instance variables are regular table fields and methods are table fields containing functions. In particular, tables have identity. That is, a table is different from other tables even if they have the same contents, but it is equal to itself even if it changes its contents over time.

One missing ingredient in the mix of tables with first-class functions is how to connect method calls with their respective objects. If `obj` is a table with a *method* `foo` and we call `obj.foo()`, `foo` will have no reference to `obj`. We could solve this problem by making `foo` a closure with an internal reference to `obj`, but that is expensive, as each object would need its own closure for each of its methods.

A better mechanism would be to pass the receiver as a hidden argument to the method, as most object-oriented languages do. Lua supports this mechanism with a dedicated syntactic sugar, the *colon operator*: the syntax `orb:foo()` is sugar for `orb.foo(orb)`, so that the receiver is passed as an extra argument to the method. There is a similar sugar for method definitions. The syntax

```
function obj:foo (...) ... end
```

is sugar for

```
obj.foo = function (self, ...) ... end
```

That is, the colon adds an extra parameter to the function, with the fixed name `self`. The function body then may access instance variables as regular fields of table `self`.

To implement classes and inheritance, Lua uses delegation [17, 18]. Delegation in Lua is very simple and is not directly connected with object-oriented programming; it is a concept that applies to any table. Any table may have a designated "parent" table. Whenever Lua fails to find a field in a table, it tries to find that field in the parent table. In other words, Lua delegates field accesses instead of method calls.

Let us see how this works. Let us assume an object `obj` and a call `obj:foo()`. This call actually means `obj.foo(obj)`, so Lua first looks for the key `foo` in table `obj`. If `obj` has such field, the call proceeds normally. Otherwise, Lua looks for that key in the parent of `obj`. Once it found a value for that key, Lua calls the value (which should be a function) with the original object `obj` as the first argument, so that `obj` becomes the value of the parameter `self` inside the method's body.

With delegation, a class is simply an object that keeps methods to be used by its instances. A class object typically has *constructor* methods too, which are used by the class itself. A constructor method creates a new table and makes it delegates its accesses to the class, so that any class method works over the new object.

If the parent object has a parent, the query for a method may trigger another query in the parent's parent, and so on. Therefore, we may use the same delegation mechanism to implement inheritance. In this setting, an object representing a (sub)class delegates accesses to unknown methods to another object representing its superclass.

For more advanced uses, a program may set a function as the parent of a table. In that case, whenever Lua cannot find a key in the table it calls the parent function to do the query. This mechanism allows several useful patterns, such as multiple inheritance and inter-language inheritance (where a Lua object may delegate to a C object, for instance).

5 Goal-Oriented Programming

Goal-oriented programming involves solving a *goal* that is either a primitive goal or a disjunction of alternative goals. These alternative goals may be, in turn, conjunctions of subgoals that must be satisfied in succession, each of them giving a partial outcome to the final result. Two typical examples of goal-oriented programming are text pattern matching [19] and Prolog-like queries [20].

In pattern-matching problems, the primitive goal is the matching of string literals, disjunctions are alternative patterns, and conjunctions represent sequences. In Prolog, the unification process is an example of a primitive goal, a relation constitutes a disjunction, and rules are conjunctions. In those contexts, a problem solver uses a backtracking mechanism that successively tries each alternative until it finds an adequate result.

A main problem when implementing problem solvers in conventional programming languages is that it is difficult to find an architecture that keeps the *principle of compositionality*. Following this principle, a piece of code that solves a problem should be some composition of the pieces that solve the subproblems. Because each subproblem may have more than one possible solution, an adequate architecture should provide an efficient way for each subproblem to produce its solutions one by one, by demand.

Lazy functional languages provide an interesting architecture for problem solving: the piece of code that solves a problem simply returns a list of all possible solutions [21]. Laziness ensures that the code actually produces only the solutions needed to find a global solution for the entire problem.

In Lua, we can use coroutines [22] for the task. A well-known model for Prolog-style backtracking is the *two-continuation model* [23, 24]. Although this model requires multi-shot continuations, it is not difficult to adapt it to coroutines that are equivalent to one-shot continuations [25, 6]. The important point is that the coroutine model keeps the principle of compositionality for the resulting system, as we will see in the following example.

Figure 2 shows a simple implementation of a pattern-matching library, taken from [6]. Each pattern is represented by a function that receives the subject plus the current position and yields each possible final position for that match. More

```
-- matching any character (primitive goal)
function any (S, pos)
  if pos < string.len(S) then coroutine.yield(pos + 1) end
end

-- matching a string literal (primitive goal)
function lit (str)
  local len = string.len(str)
  return function (S, pos)
           if string.sub(S, pos, pos+len-1) == str then
             coroutine.yield(pos+len)
           end
         end
end

-- alternative patterns (disjunction)
function alt (patt1, patt2)
  return function (S, pos)
           patt1(S, pos); patt2(S, pos)
         end
end

-- sequence of sub-patterns (conjunction)
function seq (patt1, patt2)
  return function (S, pos)
           local btpoint = coroutine.wrap(function() patt1(S, pos) end)
           for npos in btpoint do patt2(S, npos) end
       end
end
```

Fig. 2. a simple pattern-matching library

specifically, the code for a pattern yields all values j such that $\mathrm{sub}(s, i, j-1)$ (that is, the substring of s from i to $j-1$) matches the pattern.

Function `any` is a primitive pattern that matches any character. Function `lit` builds a primitive pattern that matches a literal string. Its resulting function only checks whether the substring from the subject starting at the current position is equal to the literal pattern; if so it yields the next position, otherwise it ends without yielding any option.

Function `alt` builds an alternative of two patterns: it simply calls the first one and then the second one. Each subpattern will yield its possible matchings.

Finally, function `seq` builds a sequence of two patterns. It runs the first one inside a new coroutine to collect its possible results and runs the second pattern for each of these results.

The next fragment shows a simple use:

```
-- subject
s = "abaabcda"
 -- pattern:   (.|ab)..
p = seq(alt(any, lit("ab")), seq(any, any))
seq(p, print)(s, 1)
-- results
--> abaabcda 4
--> abaabcda 5
```

It "sequences" the pattern with the `print` function, which prints its arguments (the subject plus the current position after matching `p`), and then calls the resulting pattern with the subject and the initial position (1).

6 Concurrent Programming

Traditional multithreading, which combines preemption and shared memory, is difficult to program and prone to errors [26]. Lua avoids the problems of traditional multithreading by cutting either preemption or shared memory.

To achieve multithreading without preemption, Lua uses coroutines. A *stackful* coroutine [6] is essentially a thread; it is easy to write a simple scheduler with a few lines of code to complete the system. The book *Programming in Lua* [4] shows an implementation for a primitive multithreading system with less than 50 lines of Lua code.

This combination of coroutines with a scheduler results in collaborative multithreading, where each thread should explicitly yield periodically. This kind of concurrency seems particularly apt for simulation systems and games.[2]

Coroutines offer a very light form of concurrency. In a regular PC, a program may create tens of thousands of coroutines without draining system resources. Resuming or yielding a coroutine is slightly more expensive than a function call. Games, for instance, may easily dedicate a coroutine for each relevant object in the game.

[2] Simula offered coroutines for this reason [27].

When compared to traditional multithreading, collaborative multithreading trades fairness for correctness. In traditional multithreading, preemption is the norm. It is easy to achieve fairness, because the system takes care of it through time slices. However, it is difficult to achieve correctness, because race conditions can arise virtually in any point of a program. With collaborative multithreading, or coroutines, there are no race conditions and therefore it is much easier to ensure correctness. However, the programmer must deal with fairness explicitly, by ensuring that long threads yield regularly.

Lua also offers multithreading by removing shared memory. In this case, the programming model follows Unix processes, where independent lines of execution do not share any kind of state: Each Lua process has its own logical memory space with independent garbage collection. All communication is done through some form of message passing. Messages cannot contain references, because references (addresses) have no meaning across different processes. A main advantage of multiple processes is the ability to benefit from multi-core machines and true concurrency. Processes do not interfere with each other unless they explicitly request communication.

Lua does not offer an explicit mechanism for multiple processes, but it allows us to implement one as a library on top of stock Lua. Again, the book *Programming in Lua* [4] presents a simple implementation of a library for processes in Lua written with 200 lines of C code.

The key feature in Lua to allow such implementation is the concept of a *state*. Lua is an embedded language, designed to be used inside other applications. Therefore, it keeps all its state in dynamically-allocated structures, so that it does not interfere with other data from the application. If a program creates multiple Lua states, each one will be completely independent of the others.

The implementation of Lua processes uses multiple C threads, each with its own private Lua state. The library itself, in the C level, must handle threads, locks, and conditions. But Lua code that uses the library does not see that complexity. What it sees are independent Lua states running concurrently, each with its own private memory. The library provides also some communication mechanism. When two processes exchange data, the library copies the data from one Lua state to the other.

Currently there are two public libraries with such support: LuaLanes [28], which uses tuple spaces for communication, and Luaproc [29], which uses named channels.

7 Final Remarks

Lua is a small and simple language, but is also quite flexible. In particular, we have seen how it supports different paradigms, such as functional programming, object-oriented programming, goal-oriented programming, and data description.

Lua supports those paradigms not with many specific mechanisms for each paradigm, but with few general mechanisms, such as tables (associative arrays), first-class functions, delegation, and coroutines. Because the mechanisms are not

specific to special paradigms, other paradigms are possible too. For instance, AspectLua [30] uses Lua for aspect-oriented programming.

All Lua mechanisms work on top of a standard procedural semantics. This procedural basis ensures an easy integration among those mechanisms and between them and the external world; it also makes Lua a somewhat conventional language. Accordingly, most Lua programs are essentially procedural, but many incorporate useful techniques from different paradigms. In the end, each paradigm adds important items into a programmer toolbox.

References

1. Millington, I.: Artificial Intelligence for Games. Morgan Kaufmann (2006)
2. Associação Brasileira de Normas Técnicas: Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital. (2007) ABNT NBR 15606-2.
3. Hempel, R.: Porting Lua to a microcontroller. In de Figueiredo, L.H., Celes, W., Ierusalimschy, R., eds.: Lua Programming Gems. Lua.org (2008)
4. Ierusalimschy, R.: Programming in Lua. second edn. Lua.org, Rio de Janeiro, Brazil (2006)
5. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: Lua 5.1 Reference Manual. Lua.org, Rio de Janeiro, Brazil (2006)
6. de Moura, A.L., Ierusalimschy, R.: Revisiting coroutines. ACM Transactions on Programming Languages and Systems **31**(2) (2009) 6.1–6.31
7. Ousterhout, J.K.: Scripting: Higher level programming for the 21st century. IEEE Computer **31**(3) (March 1998)
8. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The evolution of Lua. In: Third ACM SIGPLAN Conference on History of Programming Languages, San Diego, CA (June 2007) 2.1–2.26
9. Lamport, L.: LaTeX: A Document Preparation System. Addison-Wesley (1986)
10. Koyama, T., et al.: Simulation tools for damping in high frequency resonators. In: 4th IEEE Conference on Sensors, IEEE (October 2005) 349–352
11. Ueno, Y., Arita, M., Kumagai, T., Asai, K.: Processing sequence annotation data using the Lua programming language. Genome Informatics **14** (2003) 154–163
12. Kelsey, R., Clinger, W., Rees, J.: Revised^5 report on the algorithmic language Scheme. Higher-Order and Symbolic Computation **11**(1) (August 1998) 7–105
13. Nehab, D.: Filters, sources, sinks and pumps, or functional programming for the rest of us. In de Figueiredo, L.H., Celes, W., Ierusalimschy, R., eds.: Lua Programming Gems. Lua.org (2008) 97–107
14. Steele, Jr., G.L.: Rabbit: A compiler for Scheme. Technical Report AITR-474, MIT, Cambridge, MA (1978)
15. Adams, N., Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J.: ORBIT: an optimizing compiler for Scheme. SIGPLAN Notices **21**(7) (July 1986) (SIGPLAN'86).
16. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The implementation of Lua 5.0. Journal of Universal Computer Science **11**(7) (2005) 1159–1176 (SBLP 2005).
17. Ungar, D., Smith, R.B.: Self: The power of simplicity. SIGPLAN Notices **22**(12) (December 1987) 227–242 (OOPLSA'87).
18. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. SIGPLAN Notices **21**(11) (November 1986) 214–223 (OOPLSA'86).

19. Griswold, R., Griswold, M.: The Icon Programming Language. Prentice-Hall, New Jersey, NJ (1983)
20. Clocksin, W., Mellish, C.: Programming in Prolog. Springer-Verlag (1981)
21. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming **2**(3) (1992) 323–343
22. de Moura, A.L., Rodriguez, N., Ierusalimschy, R.: Coroutines in Lua. Journal of Universal Computer Science **10**(7) (July 2004) 910–925 (SBLP 2004).
23. Haynes, C.T.: Logic continuations. J. Logic Programming **4** (1987) 157–176
24. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, Snowbird, UT, ACM (September 2004) 54–65
25. Ierusalimschy, R., de Moura, A.L.: Some proofs about coroutines. Monografias em Ciência da Computação 04/08, PUC-Rio, Rio de Janeiro, Brazil (2008)
26. Ousterhout, J.K.: Why threads are a bad idea (for most purposes). In: USENIX Technical Conference. (January 1996)
27. Birtwistle, G., Dahl, O., Myhrhaug, B., Nygaard, K.: Simula Begin. Petrocelli Charter (1975)
28. Kauppi, A.: Lua Lanes — multithreading in Lua. (2009) `http://kotisivu.dnainternet.net/askok/bin/lanes/`.
29. Skyrme, A., Rodriguez, N., Ierusalimschy, R.: Exploring Lua for concurrent programming. In: XII Brazilian Symposium on Programming Languages, Fortaleza, CE (August 2008) 117–128
30. Fernandes, F., Batista, T.: Dynamic aspect-oriented programming: An interpreted approach. In: Proceedings of the 2004 Dynamic Aspects Workshop (DAW04). (March 2004) 44–50