The design and implementation of a language for extending applications

Luiz Henrique de Figueiredo, Roberto Ierusalimschy, Waldemar Celes Filho

TeCGraf–Grupo de Tecnologia em Computação Gráfica, ITS, PUC-Rio
Prédio do ITS, Rua Marquês de São Vicente 225, 22453-900 Rio de Janeiro, RJ, Brasil
{lhf,roberto,celes}@icad.puc-rio.br

Abstract. We describe the design and the implementation of Lua, a simple, yet powerful, language for extending applications. Although Lua is a procedural language, it has data description facilities, and has been extensively used in production for several tasks including user configuration, general-purpose data-entry, description of user interfaces, description of application objects, and storage of structured graphical metafiles.

Resumo. Descrevemos o projeto e a implementação de Lua, uma linguagem simples, porém poderosa, para extensão de aplicações. Embora procedural, Lua contém mecanismos para descrição de dados e tem sido largamente utilizada em produção para configuração pelo usuário, entrada de dados, descrição de interfaces, descrição de objetos da aplicação e armazenagem de *metafiles* gráficos estruturados.

Introduction

There is increasing demand for customizable applications. As applications became more complex, customization with simple parameters became impossible: users now want to make configuration decisions at execution time; users also want to write macros and scripts to increase productivity (Ryan 1990). Therefore, nowadays, larger applications almost invariably carry their own configuration or scripting languages for end-user programming. These languages are usually simple, but each has its own particular syntax. As a consequence, users have to learn (and developers have to design, implement and debug) a new language for each application.

Our first experience with a proprietary scripting language arose in a data-entry application, for which a very simple declarative language was designed (Figueiredo–Souza–Gattass–Coelho 1992). (Data-entry is an area where user defined actions are especially needed because pre-coded validation tests can hardly be adequate for all applications.) When users began to demand increasingly more power in this language, we decided that a more general approach was needed and started the design of a general-purpose embedded language. At the same time, another declarative language was being added to a different application for data description. Therefore, we decided to merge the two languages into one, and designed Lua to be a procedural language with data description facilities. Lua has since outgrown its original roots and is being used in several other industrial projects.

This paper describes the design decisions and the details of our implementation of Lua.

Extension languages

The use of languages for extending applications is now recognized as an important design technique: it allows a cleaner design for the application and it provides configuration by the user. Because most extension languages are simple, specialized to a task, they have been called "little languages" (Bentley 1986; Valdés 1991), in contrast to the "big", mainstream languages, in which applications are written. This distinction is not so sharp nowadays, since a major part of several applications is actually written using an extension language. Extension languages come in several flavors:

- *configuration languages*: for selecting preferences, usually implemented as parameter lists in command lines or as variable-value pairs read from configuration files (e.g., DOS's `config.sys`, MS-Windows' `.ini` files, X11 resource files, Motif's UIL files);
- *scripting languages*: for automating tasks, with limited flow control, such as the ones used in DOS's batch files or in the various Unix shells;
- *macro languages*: also for automating tasks, but usually only as a sequence of primitive operations, with no flow control;
- *embedded languages*: for extending applications with user defined functions based on primitives provided by the applications. These languages are usually quite powerful, being simplified variants of mainstream programming languages such as LISP and C.

What makes embedded languages different from standalone languages is that embedded languages only work *embedded* in a host client, called the *embedding* program. Moreover, the host program can usually provide domain-specific extensions to the embedded language, thus creating a version of the embedded language customized for its own purposes, possibly by providing higher level abstractions. For this, an embedded language has both a syntax for its own programs and an application program interface (API) for interfacing with hosts. Thus, unlike simpler extension languages, which are used to supply parameter values and sequences of actions to hosts, there is a two-way communication between embedded languages and host programs. Note that the application programmer interfaces with an embedded language in the mainstream language used for host programs, whereas the user interfaces with the application using solely the embedded language.

LISP has always been a popular choice for extension languages, for its simple, easily parsed syntax and built-in extensibility (Beckman 1991; Nahaboo). For instance, a major part of Emacs is actually written in its own variant of LISP; several other text editors follow the same path. However, LISP cannot be called user-friendly when it comes to customization. Neither can C and the shell languages; the latter even have a more complicated, unfamiliar syntax.

One of the fundamental decisions made in the design of Lua was that it should have a clean but familiar syntax: we quickly settled for a simplified Pascal-like syntax. We avoided a syntax based on LISP or C because it could be discouraging to outsiders or non-programmers. Thus, Lua is primarily a procedural language. Nevertheless, as already mentioned, Lua acquired data description facilities to increase its expression power.

Lua concepts

Lua is a general purpose embedded programming language designed to support procedural programming with data description facilities. Being an embedded language, Lua has no notion of a "main" program; it only works embedded in a host client (Lua is provided as a library of C functions to be linked to host applications). The host can invoke functions to execute a piece of code in Lua, can write and read Lua variables, and can register C functions to be called by Lua code. With registered C functions, Lua can be augmented to cope with rather different domains, thus creating customized programming languages sharing a syntactical framework (Beckman 1991).

This section contains a brief description of the main concepts in Lua. Some examples of actual code are included, to give a flavor of the language. A precise definition of the language can be found in its reference manual (Ierusalimschy–Figueiredo–Celes 1994).

Syntax

As mentioned before, we explicitly designed Lua to have a simple, familiar syntax. As a consequence, Lua supports an almost conventional set of statements, with implicit, but explicitly terminated, block structure. The conventional statements include simple assignment; control structures such as `while-do-end`, `repeat-until`, `if-then-elseif-else-end`; and function calls. Non-conventional statements include multiple assignment; local variable declarations, which can be placed anywhere inside a block; and table constructors, which may contain user defined validation functions (see below). Moreover, functions in Lua can take a variable number of parameters and can return many values. This avoids the need for passing parameters by reference when more than one result need to be returned.

Environment and modules

All statements in Lua are executed in a global environment. This environment, which keeps all global variables and functions, is initialized at the beginning of the embedding program and persists until its end. The global environment can be manipulated by Lua code or by the embedding program, which can read and write global variables using functions in the library that implements Lua.

The unit of execution of Lua is called a *module*. A module may contain statements and function definitions, and may be in a file or in a string inside the host program. When a module is executed, first all its functions and statements are compiled, and the functions added to the global environment; then the statements are executed in sequential order. All modifications a module effects on the global environment persist after its end. Those include modifications to global variables and definitions of new functions (a function definition is actually an assignment to a global variable; see below).

Data types and variables

Lua is a dynamically typed language: variables do not have types; only values do. All values carry their own type. Therefore, there are no type definitions in the language. The absence of variable type declarations, apparently a minor point, is actually an important factor in simplifying the language; it is frequently presented as a major feature in many variants of typed languages modified to be used as extension languages. Moreover, Lua has garbage collection: it keeps track of which values are being used and discards the ones that are not. This avoids the need for explicit managing memory allocation, a major source of programming errors. There are seven basic data types in Lua:

- *nil*: the type of a single value called `nil`;
- *number*: floating point numbers;
- *string*: arrays of characters;
- *function*: user defined functions;
- *Cfunction*: functions provided by the host program;
- *userdata*: pointers to host data;
- *table*: associative arrays.

Lua provides some automatic type conversions. A string taking part in an arithmetic operation is converted to a number, if possible. Conversely, whenever a number is used when a string is expected, that number is converted to a string. This coercion is useful because it simplifies programs and avoids the need for explicit conversion functions.

Global variables do not need declaration; only local variables do. Any variable is assumed to be global unless explicitly declared local. Local variable declarations can be placed anywhere inside a block. Therefore, because only local variables are declared, and these declarations can be made close to the use of the variable, it is usually simple to decide whether a given variable is local or global.

Before the first assignment, the value of a variable is `nil`. Therefore, there are no uninitialized variables in Lua, another major source of programming errors. Nevertheless, the only valid operations on `nil` are assignment and equality test (the main property of `nil` is to be different from any other value). Therefore, using an "uninitialized" variable in a context where an "actual" value is needed (e.g., an arithmetic expression) results in an execution error, alerting the programmer that the variable was not properly initialized. Thus, the purpose of automatically initializing variables with `nil` is not to encourage the programmer to avoid initializing variables, but rather to enable Lua to signal the use of actually uninitialized variables.

Functions are considered first-class values in Lua: they can be stored in variables, passed as arguments to other functions and returned as results. When a function in Lua is defined, its body is compiled and stored in a global variable with the given name. Lua can call (and manipulate) functions written in both Lua and C; the latter have type *Cfunction*.

The type *userdata* is provided to allow arbitrary (`void*`) C pointers to be stored in Lua variables; its only valid operations in Lua are assignment and equality test.

The type *table* implements associative arrays, that is, arrays that can be indexed with both numbers and strings. Therefore, this type may be used not only to represent ordinary arrays, but also symbol tables, sets, records, etc. To represent a record, Lua uses the field name as an index. The language supports this representation by providing `a.name` as syntactic sugar for `a["name"]`.

Associative arrays are a powerful language construct; many algorithms are simplified to the point of triviality because the required data structures and algorithms for searching them are provided by the language (Aho–Kerninghan–Weinberger 1988; Bentley 1988). For example, the core of a program that counts the occurrences of each word in a text can be written

```
table[word] = table[word] + 1
```

without having to search the list of words. (However, an alphabetically ordered report requires some real work, because the indices in a table are ordered arbitrarily inside `Lua`.)

Tables can be created in many ways. The simplest way corresponds to ordinary arrays:

```
t = @(100)
```

Such an expression results in a new empty table. The dimension (100 in the example above) is optional and may be given as a hint to the initial table size. Independently of the initial dimension, all tables in `Lua` expand dynamically as needed. Thus, it is perfectly valid to refer to `t[200]` or even to `t["day"]`.

There are two alternative syntaxes for creating tables without explicitly filling each entry: one for lists (`@[]`) and one for records (`@{}`). For instance, it is much easier to create a list by providing its elements, as in

```
t = @["red", "green", "blue", 3]
```

than with the equivalent explicit code

```
t = @()
t[1] = "red"
t[2] = "green"
t[3] = "blue"
t[4] = 3
```

Moreover, it is possible to provide user functions when creating lists and records, as in

```
t = @colors["red", "green", "blue", "yellow"]
t = @employee{name="john smith", age=34}
```

Here, `colors` and `employee` are user functions that are automatically called after the table is created. Such functions can be used to check field values, to create default fields, or for any other side-effect. Thus, the code for the `employee` record is equivalent to:

```
t = @()
t.name = "john smith"
t.age  = 34
employee(t)
```

Note that, even though `Lua` does not have type declarations, the possibility of having user functions called automatically after table creation actually provides `Lua` with user controlled *type constructors*. This non-conventional construct is a very powerful feature, and is the expression of declarative programming using `Lua`.

The application program interface

The library that implements Lua has an API, i.e., a set of C functions for interfacing Lua with host programs (there are approximately 30 such functions). These functions characterize Lua as an embedded language, and handle the following tasks: executing Lua code contained in a file or in a string; converting values between C and Lua; reading and writing Lua objects contained in global variables; calling Lua functions; registering C functions to be called by Lua, including error handlers. A simple Lua interpreter can be written as follows:

```
#include "lua.h"
int main(void)
{
 char s[1000];
 while (gets(s))
   lua_dostring(s);
 return 0;
}
```

This simple interpreter can be augmented with domain specific functions written in C and made available to Lua with the API function `lua_register`. Extension functions follow a protocol to receive and return values to Lua.

Predefined functions and libraries

The set of predefined functions in Lua is small but powerful. Most of them provide features that allow some degree of reflexivity in the language. Such features cannot be simulated with the rest of the language nor with the standard API. The predefined functions handle the following tasks: executing a Lua module contained in a file or string; enumerating all fields of a table; enumerating all global variables; type querying and conversion.

The libraries, on the other hand, provide useful routines which are implemented directly through the standard API. Therefore, they are not necessary to the language, and are provided as separate C modules, which can be linked to applications as needed. Currently, there are libraries for string manipulation, mathematical functions, and input and output.

Persistence

The enumeration functions can be used to provide persistency of the global environment within Lua, i.e., it is possible to write Lua code that writes Lua code that, when executed, restores the values of all global variables. We now show some ways to store and retrieve values in Lua, using a text file written in the language itself as the storage media. To restore values saved in this way, it is enough to execute the output file.

To store a single value with a name, the following code is enough:

```
function store(name, value)
  write(name .. '=')
  write_value(value)
end
```

Here, "`..`" is the string concatenation operator and `write` is a library function for output. The function `write_value` outputs a suitable representation of a value based on its type, using a string returned by the predefined function `type`:

```
function write_value(value)
  local t = type(value)
      if t = 'nil'    then write('nil')
  elseif t = 'number' then write(value)
  elseif t = 'string' then write('"' .. value .. '"')
  end
end
```

Storing tables is a little more complex. First, `write_value` is augmented with

```
  elseif t = 'table' then write_record(value)
```

Assuming that tables are being used as records (i.e., there are no circular references and all indices are identifiers), the value of a table can be written directly with table constructors:

```
function write_record(t)
  local i, v = next(t, nil)   -- "next" enumerates the fields of t
  write('@{')                 -- starts constructor
  while i do
    store(i,v)
    i, v = next(t, i)
    if i then write(', ') end
  end
  write('}')                  -- closes constructor
end
```

Implementation

Extension languages are always interpreted, in one way or another, by the application. Simple extension languages can be interpreted directly from source code. On the other hand, embedded languages are usually powerful programming languages, with complex syntax and semantics. A more efficient implementation technique for embedded languages is to design a *virtual machine* suited to the needs of the language, compile extension programs into *bytecodes* for this machine, and then simulate the virtual machine by interpreting bytecodes (Betz 1988, 1991; Franks 1991). We have chosen this hybrid architecture for implementing Lua; it has the following advantages over direct interpretation of source code:

- because lexical and syntactical analysis are done only once, possibly using an external parser before the actual embedding, simple errors are identified early, resulting in a shorter development cycle and faster execution;
- if an external compiler is used, there is the possibility of providing extension programs in bytecode form only, i.e., pre-compiled, resulting in faster loading, safer environments and smaller run-time support (however, linking several pre-compiled extension programs can be a difficult task).

This architecture was pioneered in Smalltalk (Goldberg–Robson 1983; Budd 1987) (from which the term *bytecodes* was borrowed) and also used in the successful UCSD Pascal system based on P-code (Clark–Koehler 1982). In these systems, bytecodes for virtual machines were used both for reducing complexity and for increasing portability. This path was also used in porting the BCPL compiler (Richards–Whitby-Strevens 1980).

Code for compilation of extension programs can be built with standard tools, such as lex and yacc (Levine–Mason–Brown 1992). The existence of good tools for compiler construction, which became widely available in the late seventies, was the main reason for the sprouting of several little languages, specially in Unix environments. Our implementation of Lua uses yacc for syntactical analysis. Initially, we wrote the lexical analyzer using lex. After performance profiles with production programs, we detected that this module was responsible for almost half of the time required for loading and executing extension programs. We then rewrote this module directly in C; the new lexical analyzer is more than twice as fast as the old one.

Lua's virtual machine

The virtual machine used in our implementation of Lua is a *stack machine*. This means that it does not have random access memory: all temporary values and local variables are kept in a stack. Moreover, it does not have general purpose registers, only special control registers, which control the stack and the execution of programs. These registers are *base of stack*, *top of stack* and *program counter*.

Programs for the virtual machine are sequences of instructions, called *bytecodes*. The execution of programs is achieved by interpreting bytecodes, each corresponding to an instruction that operates on the top portion of the stack. For example, the statement

```
a = b + f(c)
```

is compiled into

```
PUSHGLOBAL   "b"
PUSHGLOBAL   "f"
PUSHMARK
PUSHGLOBAL   "c"
CALLFUNC
ADJUST        2
ADD
STOREGLOBAL  "a"
```

Lua's virtual machine has about 60 instructions; accordingly, it is possible to use 8-bit bytecodes. Many instructions (e.g., `ADD`) do not need additional parameters; these instructions operate directly on the stack and take exactly one byte in compiled code. Other instructions (e.g., `PUSHGLOBAL` and `STOREGLOBAL`) need additional parameters, and take more than one byte. Since parameters take either one, two or four bytes, this creates alignment problems in some architectures, which are solved by padding with `NOP`s to the alignment boundary.

Many of the instructions exist for optimization only. For instance, there is a `PUSH` instruction, which takes a number as a parameter and pushes it onto the stack, but there are also single-byte optimized versions for pushing common values such as zero and one. Thus, we have `PUSHNIL`, `PUSH0`, `PUSH1`, `PUSH2`. Such optimizations reduce both the space required for compiled bytecodes and the time required for interpreting instructions.

Recall that Lua supports multiple assignment and multiple return values from functions. Therefore, sometimes, a list of values must be *adjusted*, at run time, to a given length: if there are more values than are needed, then the excess values are thrown away; if more values are needed than are present, then the list is extended with as many `nil`'s as needed. Adjustment is done on the stack with the `ADJUST` instruction.

Although multiple assignment and returns are a powerful feature of Lua, they are an important source of complexity in both the compiler and the interpreter. Because there are no type declarations for functions, the compiler does not know how many values a function will return. Thus, adjustment must be done at run time. Similarly, the compiler does not know how many parameters a function takes. Because this number may vary at run time, the list of parameters is bracketed between a `PUSHMARK` and a `CALLFUNC` instruction.

One way to extend Lua with functions provided by the host would be to assign a bytecode to each such function (Betz 1988). Although this strategy would simplify the interpreter, it has the disadvantage that fewer than 200 external functions could be added, because Lua has 8-bit bytecodes and already uses about 60 of them for its primitive instructions. We have chosen to have the host explicitly register external functions, and handling these functions like native Lua functions. Thus, there is a single `CALLFUNC` instruction; the interpreter decides what to do based on the type of the function being called.

A rather different strategy was proposed by Franks (1991): *all* external functions in the host can be called by the embedded language; no explicit registration is needed. This is done by reading and interpreting the map generated by the linker. This solution is very convenient for the application programmer, but is not portable, being dependent on the format of the map file and on the relocation strategy used by the operating system (Franks used a specific compiler for DOS).

Internal data structures

As mentioned before, variables in Lua are not typed; only values are. Thus, values are implemented in a `struct` with two fields: a type and a `union` containing the actual value. These `struct`s occur in the stack and in the symbol table, which holds all global symbols.

Numbers are stored directly into the `union`. Strings are kept in a single array; values of type *string* contain pointers to this array. Values of type *function* contain pointers to a bytecode array. Values of type *Cfunction* contain the actual pointer to the C function, as provided by the host program; the same happens for values of type *userdata*.

Tables are implemented as hash tables, with collisions handled by separate chaining (this explains why indices in a table are ordered arbitrarily). If a dimension is given when a table is created, then this dimension is used as the size of the hash table. Thus, by providing a dimension approximately equal to the expected number of indices in the table, few collisions will occur, resulting in very efficient index location. Moreover, if the table is used as an array, with numeric indices only, then choosing the right dimension at creation time guarantees that no collisions will occur.

All internal data structures in Lua are dynamically allocated arrays. When there are no more free slots in one of these arrays, garbage collection is automatically performed using a standard mark-and-sweep algorithm. If no space is recovered (because all values are being referenced), then the array is reallocated with double its current size.

Garbage collection is very convenient for the programmer because it avoids explicit memory management. When Lua is used as a standalone language (which it frequently is), then garbage collection is an asset. However, when Lua is used embedded in a host program (which is its main purpose), then garbage collection creates a new worry for the application programmer who needs to interface with Lua: care should be taken not to store Lua tables and strings into C variables, because these values may be reclaimed during garbage collection, if they do not have any further references within Lua's environment. More precisely, the programmer must explicitly copy these values into C variables, before returning control to Lua. Although this is a different paradigm, it is not worse than the familiar `malloc`-`free` protocol for memory management using the standard C library.

Conclusion

Lua has been extensively used in production since mid 93, for the following tasks:

- user configuration of application environment;
- general-purpose data-entry, with user defined dialogs and validation procedures;
- description of user interfaces;
- programmer description of application objects;
- storage of structured graphical metafiles, used for communication between graphical editors and application programs.

Moreover, Lua is currently being considered as the basis for a visual programming system.

The ability to load and execute Lua programs at run-time has proved to be a major component in making configuration an easy task for both users and developers. Moreover, the existence of a single general purpose embedded language discourages the multiplication of incompatible languages and encourages a better design, one that clearly separates the main technology contained in an application from its configuration issues.

The implementation of Lua described in this paper is available by anonymous `ftp` from `ftp.icad.puc-rio.br:/pub/lua/lua-1.1.tar.Z`.

Acknowledgments

We would like to thank the staff at ICAD and TeCGraf for using and testing Lua. The industrial applications mentioned in the text are being developed in patternship with the research centers at PETROBRÁS (CENPES) and at ELETROBRÁS (CEPEL).

References

M. Abrash, D. Illowsky, "Roll your own minilanguages with mini-interpreters", *Dr. Dobb's Journal* **14** (9) (Sep 1989) 52–72.

A. V. Aho, B. W. Kerninghan, P. J. Weinberger, *The AWK programming language*, Addison-Wesley, 1988.

B. Beckman, "A Scheme for little languages in interactive graphics", *Software, practice & experience* **21** (1991) 187–207.

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

J. Bentley, *More programming pearls*, Addison-Wesley, 1988.

D. Betz, "Embedded languages", *Byte* **13** #12 (Nov 1988) 409–416.

D. Betz, "Your own tiny object-oriented language", *Dr. Dobb's Journal* **16** (9) (Sep 1991) 26–33.

T. Budd, *A Little Smalltalk*, Addison-Wesley, 1987.

R. Clark, S. Koehler, *The UCSD Pascal handbook: a reference and guidebook for programmers*, Prentice-Hall, 1982.

M. Cowlishaw, *The REXX programming language*, Prentice-Hall, 1990.

L. H. de Figueiredo, C. S. de Souza, M. Gattass, L. C. G. Coelho, "Geração de interfaces para captura de dados sobre desenhos", *Anais do SIBGRAPI V* (1992) 169–175 [in Portuguese].

N. Franks, "Adding an extension language to your software", *Dr. Dobb's Journal* **16** (9) (Sep 1991) 34–43.

A. Goldberg, D. Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley, 1983.

R. Ierusalimschy, L. H. de Figueiredo, W. Celes Filho, "Reference manual of the programming language `Lua`", *Monografias em Ciência da Computação* **4/94**, Departamento de Informática, PUC-Rio, 1994.

J. R. Levine, T. Mason, D. Brown, *Lex & Yacc*, O'Reilly and Associates, 1992.

C. Nahaboo, *A catalog of embedded languages*, available from `colas@indri.inria.fr`.

M. Richards, C. Whitby-Strevens, *BCPL: the language and its compiler*, Cambridge University Press, 1980.

B. Ryan, "Scripts unbounded", *Byte* **15** (8) (Aug 1990) 235–240.

R. Valdés, "Little languages, big questions", *Dr. Dobb's Journal* **16** (9) (Sep 1991) 16–25.