The Lua Architecture

Advanced Topics in Software Engineering

Mark Stroetzel Glasberg

Jim Bresler

Yongmin 'Kevin' Cho

Introduction

Lua is a powerful light-weight programming language designed to extend applications.

Lua started as a small language with modest goals. The language has grown incrementally. As a result, Lua has been re-architected and rewritten several times in the past ten years. The original functional requirements and motivation of the architecture were documented in the paper "Lua-an extensible extension language" [1]. A few other versions of the language were briefly described in "The Evolution of an Extension Language: A History of Lua" [3].

The architecture and implementation was created and is maintained by Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo at the Computer Graphics Technology Group of the Pontifical Catholic University of Rio de Janeiro in Brazil (PUC-Rio).

Lua's implementation consists of a small library of ANSI C functions that compiles unmodified in all known platforms. The implementation goals are simplicity, efficiency, portability, and the ability to run on small devices with limited capabilities. These implementation goals resulted is a fast language engine with small footprint, making it ideal in embedded systems.

This paper reconstructs and documents the architecture of Lua version 5.0.2. Lua 5.0.2 contains approximately 25,000 lines of source code. The code base uses an instance of the compiler reference model, has several identifiable patterns, and is divided into clearly defined modules such as the code interpreter, parser and virtual machine.

The language is being used in several projects at Tecgraf, PUC-Rio, University of Illinois at Urbana-Champaign and in several industry companies such as Microsoft, LucasArts Entertainment, and others.

Architectural Requirements

Lua's main quality attributes, which are simplicity, performance, flexibility, and portability, were driven by the business requirements of projects developed at Tecgraf. Tecgraf is a Computer Graphics Technology Group created in May 1987 in a partnership between PETROBRAS (the Brazilian main oil company) and the Pontifical Catholic University of Rio de Janeiro - PUC-Rio. Its purpose is to develop, establish, and maintain computer graphics and user interface software for technical and scientific applications.

Business Drivers

The first project in which Lua was used needed to represent data for graphic simulators. This rudimentary ancestor of Lua was not a script language, but a data representation language. Because graphic information is naturally large, high performance was the first attribute to be considered. When users began to demand more power from this data representation language - such as the use of boolean expressions, conditional control and loops - it was clear that a true programming language was needed.

Data description played an important role in the evolution of Lua as different projects at Tecgraf were adopting Lua. Therefore, Lua had to be easily customizable for each application. Lua adopted the use of semi-structured data in the form of associative arrays. Moreover, metamethods (explained later) were introduced in the language to provide additional flexibility to data access.

Portability was an important issue for Lua. At the time, Tecgraf's main client, PETROBRAS, required all developed systems to be capable to run on a wide range of computers systems.

Finally, the business drivers also originated other functional requirements such as concurrency support, string pattern matching, garbage collection, and namespaces for modules. Most of these functionalities were added at a later point in the evolution of the language.

Quality Attributes and Tactics

As a result of its business drivers, Lua's main quality attributes were not only performance, portability, and extensibility, but also simplicity, availability, and compactness.

The development process of Lua played an important role in assuring that the quality requirements were met. New features could only be added to the language only when all committee members reached unanimity. This helped keep Lua's simplicity, which is generally considered Lua's most important asset. Lua's compactness is a consequence of its simplicity. As a result, extensibility became essential to the language.

Extensibility has been achieved by adopting several mechanisms such as data representation of C/C++ pointers (called userdata), dynamic loading of external modules, namespaces, and metamethods. Metamethods allow user-defined functions to be invoked when an associative array or a userdata variable is accessed. This allows customization of data accessing through the definition of user-defined operations. This mechanism provides flexibility to the language without making it complex.

Thanks to Lua's flexibility, several individual libraries add functionality to Lua. Some examples include: user interface support [14], socket support [15], and middleware support [16].

One tactic used to improve Lua's performance was the use of a virtual machine. A virtual machine provides a way to speed execution time because it allows precompiled code to be forwarded directly to it, therefore bypassing the lexical and syntactical analysis. This architecture was pioneered in Smalltalk (Goldberg--Robson 1983; Budd 1987) from which the term bytecode is borrowed.

Lua's portability is archived by using only ANSI C code. But even though the standard was followed closely, great care had to be taken to avoid portability problems. The heterogeneous community of users has been helpful identifying compilers discrepancies and solving compiling issues throughout the language's evolution.

Availability has been achieved as a consequence of compactness, the use of good implementation standards, and a big set of tests to catch bugs.

Lua Features

Lua offers a wide variety of programming functionalities such as loops, scope, function calls, arithmetic calculations, string pattern matching, error handling (exception like), coroutines, garbage collection, debugging mechanisms, comprehensive C API, OS facilities, input and output functions, and more.

Following its objective of simplicity, Lua defines a reduced number of data types. Although the number of data types is reduced, they are powerful enough to accomplish all of Lua's objectives. Below we present and describe Lua's data types.

1. **string** represents arrays of characters. Lua is 8-bit clean: Strings may contain any 8-bit character, including embedded zeros ('\0').
2. **number** is defined by default at compile time as the *double* type in C. This simplifies the use of number types in Lua by allowing the user to represent just about any number: integer, positive, negative, and floating point. In addition, user may change the number type of Lua in compile time.
3. **nil** is a special empty value.
4. **boolean** is defined as true, or false.
5. **function** is a first-class value. It can be either a Lua or C function and can be stored in variables.
6. **userdata** allows arbitrary C data to be stored in variables.
7. **thread** represents independent thread of execution and is used to implement coroutines.
8. **table** is the fundamental data structure mechanism defined by Lua. Lua tables are associative arrays that allow the user to index and store any other Lua data type. This allows the user to create many types of data structures such as lists, trees, and graphs.

One important design consideration is Lua's ability to classify and represent data. Lua is a weakly typed language; this means that variables do not have types, but the variable's values do.

Among Lua's data types are tables and userdata. Lua uses the table and userdata types to support application specific structures. Tables are really associative arrays [6], an abstract data type that behaves similarly to arrays but allows anything to be a key or a value. This structure allows several data structures to be represented such as trees, graphs, or even XML files.

In addition, Lua allows the semantics for both userdata accesses and table accesses to be altered through metamethods. Metamethods enable applications to override the default logic of data access, supporting for example the implementation of object oriented mechanisms [18].

Changing table access policies enables the user to reflect a C++ variable in Lua and translate Lua method calls to C++ method calls. This mechanism is used extensively by tolua [13], a tool that helps creates C and C++ bindings to Lua. Finally, Lua allows metamethods to be changed during execution, adding even more flexibility to the mechanism.

Another characteristic of Lua function calls is that function parameters are passed as references, so function calls are fast. Nonetheless, parameters cannot be changed inside functions; as a result, the number of function return values becomes limited. Lua solves this issue by allowing multiple values to be returned in function calls.

Architectural Solution

In this section we describe the architecture of Lua. This text describes first how Lua receives and interprets a script file. Next, we present the internal module decomposition of Lua and how each module interacts with each other. Finally, we describe most of Lua's subsystems.

Lua: An Embedded Script Language

Although Lua offers a stand alone command line interpreter, Lua is designed to be embedded in an application. Applications can control when a script is interpreted, loaded, and executed. They can also catch errors, handle multiple Lua contexts, and extend Lua's capabilities.

The process of initializing Lua and loading a script is depicted in Figure 1.

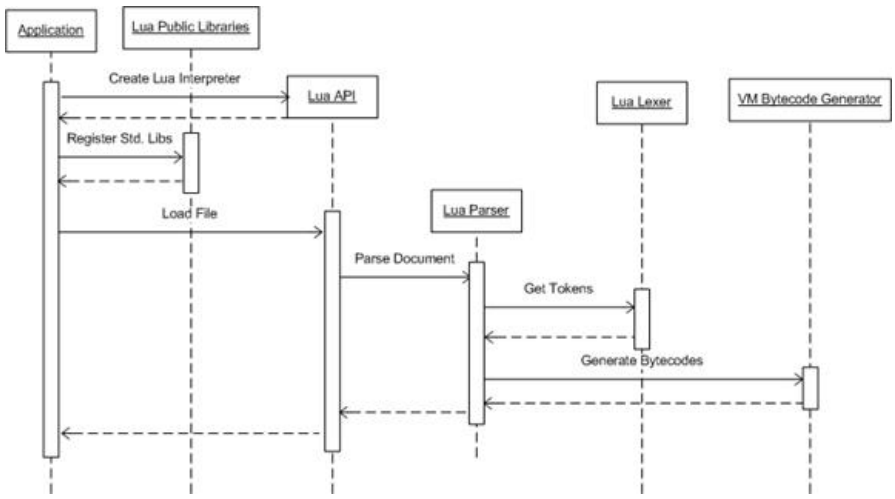

Figure 1: process of initializing Lua and loading a script file

Four steps are necessary to load and execute a Lua script. First, a state of the Lua interpreter must be created. This state is passed on to all functions of Lua's C API, including the calls done in the following steps. Second, the application embedding Lua registers all libraries that extend Lua. Next, scripts provided by the application are parsed and instructions that the virtual machine can execute are generated. These instructions are referred to as bytecodes. Finally, the bytecodes are forwarded to the virtual machine for execution.

The first step to loading and executing a script is creating a Lua interpreter reference. This step consists of initializing a lua_State structure by calling lua_open. The lua_State structure is necessary because Lua offers a reentrant API and does not use any global variable. As a result, an application may create multiple instances of the Lua interpreter.

Next, the application needs to register libraries available to Lua programs. Lua supports a default set of libraries for target applications. Applications may expand or contact the list of libraries available to Lua programs by controlling which libraries are registered. This allows applications to customize the library functions available to Lua applications.

Afterwards, the Lua interpreter needs to obtain bytecodes to execute. At this point, there are two possible scenarios: precompiled Lua bytecodes are loaded or a Lua script is loaded. When loading a script, Lua uses standard lexer, parser, and bytecode generation components to precompile the program. These components behave like Pipes and Filters [11] by passing data to each other sequentially and incrementally. Because each of these components has a significant impact on performance, Lua needs to execute these components as quickly as possible. Therefore, Lua does not use automated code generation tools such as lex or yacc [17]; instead, the Lua implementation has a hand-written parser and lexer.

Finally, Lua needs to execute the bytecodes. The virtual machine kernel contains a loop that reads and executes a virtual machine instruction.

Module Decomposition

Lua is divided in subsystems to support its requirements. These modules include an application loader, library loader, a public API, auxiliary libraries, several modules to support the virtual machine, and several modules to support translating Lua script into bytecode. A static view of Lua's modules and relationships between them is depicted in Figure 2. In addition, the picture shows the implementation file of each module.

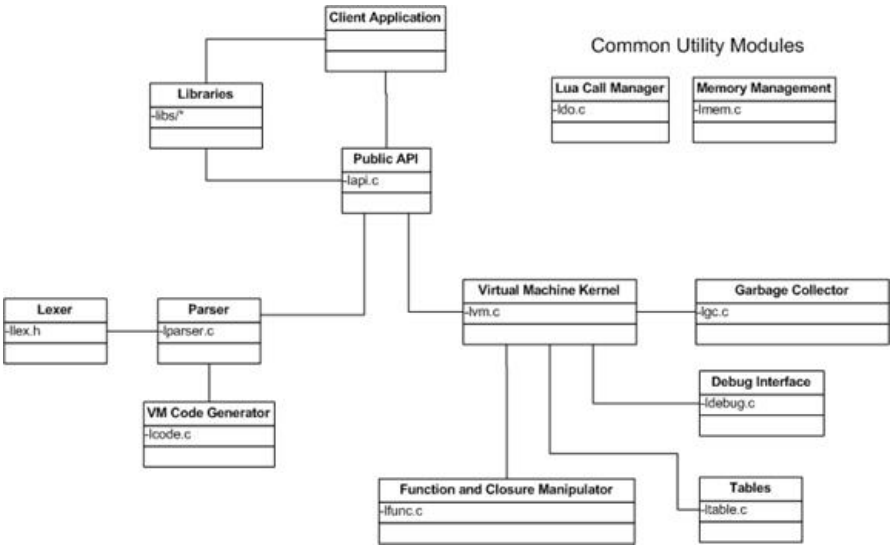

Figure 2: Lua Module Decomposition

Lua's module decomposition helps Lua address its quality and functionality requirements. In particular, the module decomposition helps Lua maintain its compactness goals because it allows detachment of modules from the normal distribution. For example, Lua separates core platform code and auxiliary code by placing them into different libraries. As a result, applications embedding Lua are not required to link the auxiliary library. The module decomposition also allows applications to reduce Lua's footprint by removing the parser subsystem if the application only needs to execute precompiled programs.

Another advantage of Lua's module decomposition is that it minimizes the dependencies between external applications and Lua. The public API presents a facade [11] that allows applications to use Lua without knowledge of its internal decomposition. This allows the Lua team to make several types of changes without breaking compatibility with existing applications. As a result, applications can often upgrade the version of the Lua interpreter without making any source code changes.

However, the public API is difficult to use. The auxiliary library exposes a simplified API to applications that is easier to use than the public API. For example, the auxiliary library provides functions that load a Lua script or precompiled file from disk. The auxiliary library is powerful enough to address the requirements of most applications that embed Lua. However, applications still have the ability to use the core API when the auxiliary library is not sufficiently powerful.

The public API interacts with subsystems to perform operations requested by the application. For example, the public API interacts with the parser subsystem to convert Lua scripts to bytecodes. The parser subsystem consists of a lexer, parser, and bytecode generator.

Another important subsystem is the virtual machine. The virtual machine helps Lua achieve its performance requirements. Its modules consist of a virtual machine kernel, garbage collector, debugging interface, and others.

The virtual machine increases performance by decoupling language syntax from application execution semantic, allowing faster loading of pre-compiled scripts. For this purpose, the Lua distribution offers an external compiler that allows translation of Lua scripts to bytecode form. This bytecode form also serves as code scrambling, hiding the code source in the application's final distribution.

Lua also uses several utility modules to meet its requirements. For example, Lua has an error detection module that allows errors to be handled in a centralized manner. The module allows the execution of C function in *protected mode*. A function running in protected mode can throw an exception if it encounters an error. When a function throws an exception, control returns immediately to the caller of the protected mode function.

Subsystems

Lua is divided into subsystems that separate functionality. We have explained how the subsystems work together; now we explore in more detail some of Lua's subsystems.

Parser

The objective of the Lua parser subsystem is to convert Lua scripts to bytecodes that the virtual machine will eventually execute. Therefore, the parser subsystem consists of a lexer, a parser, and a bytecode generator.

The lexer and parser follow the compiler reference model. In this model, the lexer is responsible for obtaining tokens, which are separate identifiable parts of the script file such as constants, operators, keywords, and others. The parser is responsible for analyzing the structure in which the tokens are disposed in the file in order to compose commands. In addition, the parser can generate error messages for invalid command constructions.

The Lua parser uses only one-pass to maximize performance. This is viable because Lua does not perform type checking because all variables are untyped. In addition, function existence and prototype verification of calls are not checked during

parsing. However, this solution causes problems during the programming process because many errors are only caught during run-time.

The parser subsystem defines the concept of a chunk. A chunkis a unit of execution which consists of a series of command statements. The parser is designed to work on one chunk at a time. Chunks can be provided by the host program in the form of a file or a string. When Lua script is supplied incrementally (for example by providing a sequence of separate strings to be interpreted), a new chunk is created every time. In each of the chunks the user can specify direct statements, functions, local variables, and return values.

The parser will generate bytecode for each chunk that it interprets. Chunks can then be passed on to the virtual machine for execution. This can be passed directly through memory or each chunk can be persisted to disk as precompiled bytecodes for later execution in the virtual machine.

Virtual Machine

The virtual machine subsystem is responsible for executing bytecodes generated by the parser subsystem. Because most processor time is spent in the virtual machine subsystem after the Lua script is loaded, the design of the virtual machine subsystem has a large impact on overall system performance. Furthermore, the virtual machine subsystem cannot be removed from a minimal Lua interpreter. As a result, the virtual machine subsystem needs to be compact and have good performance.

The virtual machine kernel can read and execute bytecodes. To support this functionality, the virtual machine kernel continuously loops for the next operation to execute. The loop identifies the type of operation and performs an instruction-specific task to execute it. Several modules, such as the garbage collector, table, and closure modules, are invoked by this loop to assist the virtual machine kernel.

Bytecode operations have been carefully designed to ensure subsystem compactness and performance. All instructions are 32-bit long and can contain from one to three arguments. The arguments are typically denoted as A, B, and C. Some instructions combine the B and C into a bigger argument called Bx. All opcodes have a constant length.

The virtual machine organizes the data that it must handle in a global table, a local constant table, and several registers. The purpose of the global table is to store values that be accessed anywhere in a Lua script. The constant table stores constant data (such as strings, numbers, etc.) so that virtual machine operations do not need to directly use constant values. Data in the constant table is indexed by position. Multiple references to the same constant have the same position in the constant table.

Stacks are used during function calls and are created for each closure. A closure is basically an instance of a function. Local variables defined in a closure reside in the closure' s corresponding stack for the duration of the closure. Moreover, designated variables that reside in the outer scope (usually called upvalues) can be accessed. When the function is completed, the closure is finished and the corresponding stack is cleaned. In the next section we explain some of the problems that need to be addressed when a function finishes.

Finally, in appendix A we describe two examples of how a Lua script is translated into virtual machine instructions. Additional information about the Lua virtual machine and its bytecodes is presented in the document "A No-Frills Introduction to Lua 5 VM Instructions" [20].

Closures

Lua defines the scope of variables as global or local. Global variables can be accessed anywhere during the execution of a script. The scope of local variables begins at the first statement after their declaration and lasts until the end of the innermost block that includes the declaration. Variables that are in the outer scope can always be accessed from the inner scope.

Functions in Lua are first class values. This means that functions can be assigned to variables just like any other value. In addition, functions can be defined inside other functions and returned as values. As a result, nested functions can be executed even after their containing block has finished executing. Because the nested function can access data outside of its

local scope, Lua cannot discard a block's variables after it finishes executing.

Usually, a stack holds local variables; when the function call ends, the values are removed from the stack and destroyed. Because of Lua's scope visibility rules, these variables cannot be immediately released. Lua solves the problem by using a stack to maintain local variables when the function is executing. In addition, Lua keeps a linked list of pending variables that will be checked for possible references after the block executes. After the block executes, Lua checks which local variables are still needed. Finally, variables that are still referenced are moved from the stack to the heap.

The left side of Figure 3 presents the internal Lua organization during a function call. Both nested and enclosing functions access local variables from the stack to guarantee consistency. The right side of Figure 3 shows what happens after the function ends: the stack removed the function variable and moved it to the heap. This allows more than one nested function to access external local variables of outer scopes.

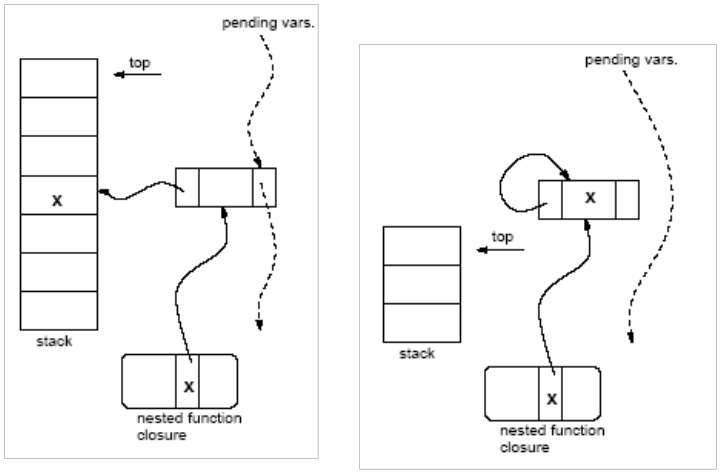

Figure 3: Lua Closures

The fact that Lua allows local function variables to persist through nested functions (even after the outer function has ended) provides Lua with an additional benefit. Lua can act in an object-oriented style by allowing applications to restrict variable accesses to specific functions.

Garbage Collector

Lua uses a generic mark and sweep algorithm to implement garbage collection. The amount of data consumed during the execution of a Lua script is tracked. When a threshold value is reached, the garbage collector frees all data that is no longer referenced. However, this approach has a drawback. The garbage collection operation can occur at any time, creating problems when real-time applications use Lua. Furthermore, the garbage collector is executed very frequently for applications that use a lot of memory. For these reasons, major changes are expected for the garbage collector in Lua version 5.1.

Table

Tables are the only data structure in the Lua language. Tables are a powerful construct that maintains language simplicity and generality. Internally, a table is implemented as two separate parts: a hash table and an array.

Non-negative integer keys are all candidates to be kept in the array part. The actual size of the array is the largest index value such that at least half the slots between zero and this value are in use.

The hash function uses a chained scatter table with Brent's variation [21]. If a table element is not in its main position (i.e. the

position given by the hash function) then a colliding element must be in the objects main position. In other words, there are collisions only when two elements have the same main position. Because of this, the load factor of these tables can be 100% without performance penalties.

The data structure is very efficient. Nevertheless, like all hash tables, its performance depends on the distribution of hashes. So far, the Lua community has not reported any problems with string hashes.

Figure 4 shows an example of a Lua table which is composed of a hash table and an array. As shown in Figure 4, the fifth numeric value of the array can either be stored in the hash table or in the array. Numeric keys which do not fall within the capacity of the array part are handled normally. This mechanism is transparent to other Lua modules.

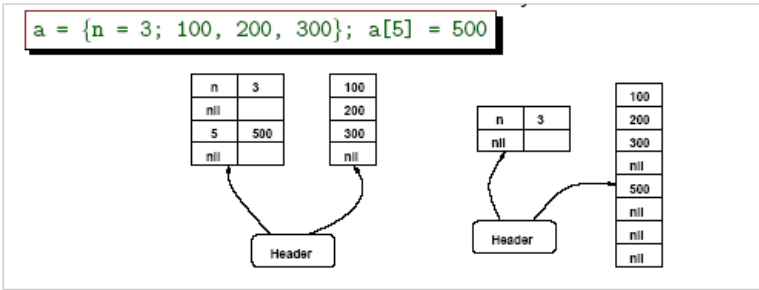

Figure 4: Example of a Lua table internal organization

Standard Library

Lua includes a standard library containing functionality needed by several applications such as string pattern matching, file I/O, and basic math support. In addition, the standard library interfaces with the Lua runtime to help support several language features such as coroutines and tables. For more information, see the Lua reference manual [7].

Dynamic Libraries

Dynamic Library support provides a key mechanism to extend Lua. It allows applications that embed Lua to expand and reduce the functionality available to scripts. As a result, dynamic libraries help Lua achieve its compactness and flexibility requirements.

Applications embedding Lua initiate the process to load libraries. Libraries registered by the embedding application can be linked statically or dynamically. This allows applications to link only the subset of libraries it needs. Usually, the application initiates the extension library by calling an initialization function that registers global tables, variables and/or functions in the Lua' s script environment.

Dynamic libraries can also be loaded directly from the Lua environment. This is done by calling the loadlib function with the filename of the dynamic library and the name of its initialization function. The use of this function allows the user to locate the library and call its initialization function.

Lua' s library architecture also enables loading parts of a library. For instance, the standard Lua functions include separate calls to initialize the coroutine library, the auxiliary library, string matching library, library loader, etc. This allows application greater control over what functions are available to script it embeds.

As an example, consider an application that wants to allow its script to access every part of the standard Lua libraries except the dynamic library loader. The application should call the every library initialization in the Lua standard library except luaopen_loadlib.

Coroutines

Lua supports asymmetric coroutines [19]. The virtual machine kernel is responsible for maintaining state associated with the virtual machine subsystem. State managed by the virtual machine kernel consists of one or more execution states, each of which can fully specify the state of the processor. This design allows coroutines to be implemented by creating new execution states each time a new coroutine is loaded and the active execution state is changed when a coroutine yields.

Conclusion

Lua has archived its goal of simplicity, extensibility, portability, reliability, and performance.

Most goals derive from the fact that the language is simple. Simplicity was archived by allowing the user to create language features through the use of meta-mechanisms instead of implementing features directly in the language. Meta-mechanisms allow users to implement inheritance, namespace, and debugging. Other features such as type checking are not included in Lua.

Extensibility was accomplished by a comprehensive interface between Lua and C/C++. The API allows Lua to call functions in C, create objects in C++, and call methods in C++ objects. Furthermore, C/C++ applications can easily call Lua functions and access data in variables, tables, and metamethods. The success of extending the Lua library is demonstrated by the number of libraries that extend Lua's functionalities [13,14,15,16].

Reliability was achieved by extensively using tests to certify language stability. As described in History of Lua [3], "to keep a language is much more than to design it. Complete attention to detail is essential in all aspects." In addition, reliability was achieved by using an established reference model of a compiler.

The Lua architecture places a significant emphasis on performance. Performance tactics include using a virtual machine that can receive precompiled opcodes and implementing the lexer and parser by hand. The virtual machine allows applications to completely bypass the normal parse sequence when the Lua scripts have been precompiled. Implementing the lexer and parser by hand improves performance when a script is loaded by providing more efficient components than automated tools could provide.

Appendix A: VM Code Generation Examples

The virtual machine supports 51 operations, although most programs use a small subset of these operations. Lua VM operations can be generated in a human readable form by running the external bytecode generator (called luac) with the "-p" and "-l" options.

First, consider the simple Lua script below:

```
y = 5
print(y)
```

Running this script through luac with the "-p" and "-l" generates the following bytecodes:

```
LOADK          0 1    ; 5
SETGLOBAL      0 0    ; y
GETGLOBAL      0 2    ; print
GETGLOBAL      1 0    ; y
CALL           0 2 1
RETURN         0 1 0
```

The LOADK operation loads a value from the constants table and places it in a register. Its first argument is the stack register to store the result in and its second argument is an index into the global constants table. The GETGLOBAL operation reads a value from the globals table. Its result is stored in the register specified by the first argument. The second argument is a reference to a constant table entry that has a pointer to a global value. The SETGLOBAL operation sets a value in the globals table. Its first argument is a register that has the value to be saved in the globals table. The second argument is an index into the constant table that has a pointer to a value in the globals table. The CALL operation calls a function. Its first argument gives the register that has a pointer to the function. The function arguments are placed sequentially in registers after the function pointer. Its second argument contains the number of function arguments plus one. Its third argument contains the number of values that the function returns. The return values are placed sequentially in the registers starting at the register specified for the first argument. Finally, RETURN is called when a function is finished. The first argument is the stack position of the returned first value; the other returned values are in the following positions. The second argument of the RETURN statement is the number of returned values plus one.

In the example, the first two operations are responsible for setting y to 5. The first operation loads constant #1 and places it in register #0. The second operation takes register #0 and places it in a list of globally accessible values indexed by constant #0. From these opcodes, we can determine that constant 0 contains the global value index to y and constant 1 contains the number 5.

The next four operations of the example are used to call the print function. The first GETGLOBAL operation retrieves the print function from the global table and places it in register #0. The second operation places y in register #1. Finally, we execute the CALL operation. The CALL operation expects the called function and all of its arguments to be in sequential registers. Because the print function is stored in register #0 then the first argument must be stored in register #1. The third argument of the CALL function is the number of expected return values plus one.

Now, let's consider a more complex example:

```
local x = 10
function test()
  x = 2
end
```

luac generates the following bytecodes for this example:

```
main:
    LOADK          0 0    ; 10
    CLOSURE        1 0    ; 0xa051940
    MOVE           0 0 0
    SETGLOBAL      1 1    ; test
    RETURN         0 1 0
```

```
test:
    LOADK       0 0    ; 2
  SETUPVAL      0 0 0  ; x
  RETURN        0 1 0
```

This example uses three opcodes that the previous example did not: CLOSURE, SETUPVAL, and MOVE. The objective of the CLOSURE instruction is to create an instance of a function. Its first argument specifies the base register that the instantiated function references. The base is used to store a reference to the instantiated function. If the instantiated function has upvalues, one register for each upvalue will be reserved sequentially created after the base register. The CLOSURE operation's second argument is the index into the table of function prototypes of the function we want to access.

The SETUPVAL instruction sets the value of a variable in the outer scope. Its first argument is the register that has the data source. The second argument is the destination's index in the upvalues list.

The MOVE instruction copies data from a specific register to another. The first argument is the destination register and the second argument is the source. The third argument is ignored.

In the provided example, the first operation of the main chunk sets the x variable to 10 by loading constant #0 into register #0. The second operation places a reference to the function testfunc in register #1. The third operation has no affect because the first and second arguments of the move operation refer to the same register. Next, the fourth operation stores a reference to the function *test* in the global table so it can be called outside the current scope. Finally, the last instruction in the main chunk is a return instruction which effectively ends the chunk.

The second part is the code of the test function. The first operation in test loads 2 into register #0 by copying the value stored in the local constant's table. Next, the test function sets *x* to 2 by setting the upvalue #0 to the value in register #0. Finally, the test function returns control to the previous caller.

Acknowledgements

We would like to thank all the authors of Lua, in particular Roberto Ierusalimschy who suggested the description of some features, provided reference material for the construction of this document, and gave us figures for closures and tables.

In addition, we would like to thank Ralph Johnson for assisting on organizational aspects of this document, as well as, the main content definition.

Moreover, Rici Lake's email about the hash table implementation has supplied some of the content of the Table section.

References

[1] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, "*Lua-An Extensible Extension Language*", Software: Practice & Experience26 pp 635-652 #6 (1996).

[2] L. H. de Figueiredo, R. Ierusalimschy, W. Celes, "*The Design and Implementation of a Language for extending applications*", Proceedings of XXI Brazilian Seminar on Software and Hardware, pp 273-283 (1994).

[3] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, "*The Evolution of an Extension Language: A History Of Lua*", Proceedings of V Brazilian Symposium on Programming Languages B-14--B-28 (2001).

[4] L. H. de Figueiredo, R. Ierusalimschy, W. Celes, *"Lua: An Extensible Embedded Language a Few Metamechanisms Replace a Host of Features"* ,Dr. Dobb's Journal21 #12 pp 26-33. [DDJ] (Dec 1996)

[5] J. Gracia-Martin, M. Sutil-Martin, *"The Extreme Compiling Pattern Language"*

[6] Associative Array at Wikipedia, "*http://en.wikipedia.org/wiki/Associative_array*"

[7] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *"Lua 5.0 Reference Manual"* , Technical Report MCC-14/03, PUC-Rio, 2003

[8] L. H. de Figueiredo, R. Ierusalimschy, W. Celes, *"The Design And Implementation Of A Language For Extending Applications"* , Proceedings of XXI Brazilian Seminar on Software and Hardware pp 273-283 (1994).

[9] Len Bass, Paul Clements, and Rick Kazman; *"Software Architecture in Practice"* , second edition, Apr 9, 2003, Addison-Wesley

[10] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stahl; "*Pattern-Oriented Software Architecture: A System of Patterns*" 1996, Wiley & Sons

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; "*Design Patterns*" 1994, Addison-Wesley

[12] "*The Kepler Project*" at http://www.keplerproject.org, 2004, Fábrica Digital and PUC-Rio

[13] "*tolua*" at http://www.tecgraf.puc-rio.br/~celes/tolua/ , 2004, Tecgraf/PUC-Rio

[14] "*IUP*" http://www.tecgraf.org/iup, 2004, Tecgraf/PUC-Rio

[15] "*LuaSocket*" at http://www.tecgraf.org/luasocket, 2004, Fábrica Digital and PUC-Rio

[16] "*Luaorb*" at http://www.tecgraf.puc-rio.br/luaorb , 2004, Tecgraf/PUC-Rio

[17] John R. Levine, Tony Mason, Doug Brown; "*Lex & Yacc*" 1992, O'Reilly & Associates

[18] David Jeske, "*A Fast Multiple-Inheritance Tag Method Implementation in Lua*", at http://www.lua.org/notes/ltn008.html Lua Technical Note 8.

[19] Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy; *"Coroutines in Lua"* July 2004 Journal of Universal Computer Science, pp 910-925 10(7).

[20] Kein-Hong Man. "*A No-Frills Introduction to Lua 5 VM Instructions*" , November 28th, 2004

[21] Richard P. Brent; *"Reducing the retrieval time of scatter storage techniques"*