Programming in Lua – The Lua Implementation

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua
Navigating the source

- The source code for Lua 5.2 is online at [http://www.lua.org/source/5.2/](http://www.lua.org/source/5.2/)

- *Includes* lists the three include files that external libraries use, plus *luaconf.h*, for compile-time configuration of Lua.

- *Core* lists the files that implement the Lua compiler and virtual machine.

- *Libraries* is the code for the built-in functions and modules of the standard library, all implemented in terms of the C API.

- *Interpreter* is actually just the REPL, the hard work is done by the core; the REPL just uses API functions!

- *Compiler* is also just a shell around the actual compiler that is in the core.
A quick tour of the core

- `lapi.c` implements the C API (functions with `lua_` prefix); the `luaL_` API functions are actually in `lauxlib.c`!

- `lobject.h` has the representation of Lua values

- `lstate.h` has the (internal) representation of Lua states, private to the core

- `lpcodes.h` has the instruction format and the list of instructions for the virtual machine

- `lvm.c` is the core of the virtual machine, with its execution loop (in `luaV_execute`) and some support functions
A quick tour of the core (2)

- `ldo.c` implements function calls and the management of the call stack and the value stack, as well as error handling.

- `lstring.c` manages the “string table”, where Lua keeps a canonical copy of each string; the actual string values are just pointers to entries in this table.

- `ltable.c` is the implementation of tables, and has the logic for handling the table’s array and hash parts, and resizing.

- `ltm.c` has a few functions to fetch metamethods (they were called *tag methods* prior to Lua 5.0).

- `lfunc.c` has a few functions to handle prototypes (the code for a function) and closures.
A quick tour of the core (3)

- `debug.c` has the functions of the debug API, and their support functions

- `gc.c` is the garbage collector, managing the memory used by Lua and freeing memory when it is not used anymore

- `ldump.c` and `lundump.c` handle VM instruction serialization and deserialization

- `parser.c` and `lcode.c` are the recursive descent parser and the code generator for the Lua compiler

- `lex.c` is the scanner for the compiler; the scanner and deserializer both use the stream interface in `lzio.c` to get the bytes they need
The Lua scanner

- Lua has a simple lexical structure, and uses a hand-written scanner

- The scanner itself has some complexity due to it having to interface with the stream interface, the memory manager, and the string table

- We do not actually need to change the source code for the scanner to do some simple changes

- We have some simple hooks into the scanner in the form of lis* macros that it uses to classify a byte as a digit, alphabetic, alphanumerical, or space character
UTF-8 identifiers

- We can use the hooks in to the scanner to add support for UTF-8 identifiers

- We just change the definitions of some of the macros in `lctype.h`

```c
/*all utf-8 chars are always alphabetic character (everything higher then 2^7 is always a valid char), end of stream (-1) is not valid */
#define lislalpha(c) (((0x80&c)||isalpha(c))&&c!=-1)
/*all utf-8 chars are always alphabetic character or numbers, end of stream (-1) is not valid*/
#define lislalnum(c) (((0x80&c)||isalnum(c))&&c!=-1)
```

```lua
function 提出反()
  local n = 0
  return function ()
    n = n + 1
    return n
  end
end

计数器 = 提出反()
print(计数器())  -- 1
print(计数器())  -- 2
print(计数器())  -- 3
```
The Lua parser

- Lua uses a hand-written recursive parser; basically, each grammar rule corresponds to a function in the parser, beginning with `statlist` for a list of statements.

- But the parser is greatly complicated by the fact that the parser is generating code as it goes, instead of first building an intermediate representation.

- The exception is the expression parser, a precedence climbing parser that generates an abstract syntax tree for expressions.

- The code generator for expressions traverses this tree.
Values

- Lua values are *tagged unions*: a structure containing a *tag* for the value (the type plus some bookkeeping information for the VM) and an union with fields for each kind of value:

  ```c
  union Value {
    GCObject *gc; /* collectable objects */
    void *p; /* light userdata */
    int b; /* booleans */
    lua_CFunction f; /* light C functions */
    numfield /* numbers */
  }
  ```

- GCObjects are strings, tables, functions, threads, and userdata; all types that have memory managed by the Lua garbage collector

- Plus some internal values that the VM uses: upvalues and prototypes
GCObjects

- The common header is duplicated in all of the different GCObject parts, and is bookkeeping information for the garbage collector:

```c
union GCObject {
    GCHeader gch; /* common header */
    union TString ts;
    union Udata u;
    union Closure cl;
    struct Table h;
    struct Proto p;
    struct UpVal uv;
    struct lua_State th; /* thread */
};
```

- Notice that threads are just Lua states; the difference is that they have a link to, and share global variables with, their parent Lua state
Tables

• Tables have an *array* part and a *hash* part (the array of nodes in node, below):

```c
typedef struct Table {
    CommonHeader;
    lu_byte flags;  /* 1<<p means tagmethod(p) is not present */
    lu_byte lsizenode;  /* log2 of size of `node' array */
    struct Table *metatable;
    TValue *array;  /* array part */
    Node *node;
    Node *lastfree;  /* any free position is before this position */
    GCObject *gclist;
    int sizearray;  /* size of `array' array */
} Table;
```

• Notice that the fact that a metatable *must* be another table is fixed in the implementation
Tables – hash part

• Each node in the hash part has a *key*, a *value*, and a link that is used for collision resolution in the hash table.

• Lua uses a hash algorithm that can handle a close to full table quite well, so the hash table only grows when it runs out of space.

• Each time the hash part grows it doubles in size.
Tables – array part

- Lua tries to keep as many values with integer keys as it can in the array part of the table, without wasting much space.

- Each time the table rehashes, Lua sets the array part to size $n$, where $n$:
  - Is a power of 2
  - Contains at least $n/2$ values in the interval $[1, n]$, that is, is at least half full
  - Has at least one value in $[n/2 + 1, n]$, that is, it is not wasting the upper half
  - Rehashing is an expensive operation, but the doubling in size of each part makes it infrequent
Virtual Machine

• Lua has a *register-based* virtual machine
  
• Each Lua function gets a number of *virtual registers*; it will have one for each argument, usually one for each local variable, and how many it may need to keep temporary values
  
• Makes for very compact code, and a large number of virtual registers simplifies code generation, there is no need for “register allocation” in the Lua compiler
  
• Instructions can take up to three registers, although some of them operate on *ranges* of registers
  
• The second and third operands can also be *constants*, which are indexes on an array of literals that each function has
Examples

- In the instructions below, registers are given by $R^n$, where $n$ is the register number, and numbers are indexes in the array of constants:

  ADD $R0$ $R0$ 3
  DIV $R0$ 3 $R0$
  GETTABLE $R0$ $R1$ 4
  SETTABLE $R0$ $R1$ 4

- If we assume that $R0$ is the local variable $a$, $R1$ is the local variable $t$, constant 3 is the number 1, and constant 4 is the string “x”, then the above corresponds to the Lua code:

  ```lua
  a = a + 1
  a = 1 / a
  a = t.x
  t.x = a
  ```

- Sometimes the second and third operands are neither registers nor constants; the “register” is just an integer:

  ```lua
  NEWTABLE $R1$ $R0$ $R0$ ; t = {}
  ```
“Large” operands and tests

- A second instruction format takes just two operands, where the second can be a large number (usually for a jump offset, but it can also be an index in the array of literals):

  LOADK R0 1000 ; assigns literal with index 1000 to the first register
  JMP R0 -500 ; jumps backwards (ignores the first operand)

- Tests have a dummy first argument that is either R0 or R1 and gives the “polarity” of the test; R0 makes it skip the jump if the test succeeds, and R1 makes it skip the jump if the test fails:

  LT R0 R0 3 ; if a < 1 then a = a - 1 else a = a + 1 end
  JMP R0 2 ; jumps 3 instructions ahead
  SUB R0 R0 3
  JMP R0 1 ; jumps 2 instructions ahead
  ADD R0 R0 3
Prototypes and closures

• The Lua compiler produces a prototype for each function

• The prototype has the instructions for the function, and metadata used by the virtual machine:
  • How many registers the function uses
  • In which source file and at which line the function comes from
  • Which local variables from outside its scope the function uses

• A function declaration becomes a CLOSURE instruction, which creates a closure from the prototype
Creating a closure

• When the virtual machine creates a closure, it uses the list of external variables to fill the closure’s *display*.

• The display is an array of *upvalues*, one for each external variable the function uses.

• Upvalues may be *open* or *closed*; an open upvalue means that the variable is still in scope, and points the the location of the variable in the *stack*.

• A closed upvalue means the variable has gone out of scope, and now holds the value the variable had.
Closures and sharing

• Two or more closures may share a local variable, so each variable must have at most just one open upvalue pointing to it

• Lua keeps the implementation simple by maintaining a linked list of open upvalues, and searching this list each time it needs to create a closure

• If no open upvalue for a variable is found, Lua creates one and adds it to the list

• When an upvalue is closed it is removed from the list

• Each time a block goes out of scope the Lua compiler generates code to close any open upvalues in it, using the first argument of the JMP instruction
Lua assembler/disassembler

- luaa.lua and luad.lua are two Lua scripts that let us experiment with programming directly to the Lua VM

- One is an **assembler**, to turn textual instructions into executable code, and the other is a **disassembler**, to turn Lua code into textual instructions:

  ```bash
  $ lua luad.lua -o test.asm test.lua
  $ cat test.asm
  function main(θ):
    .upvalue _ENV, 1, 0
    1 [1] LOADK R0, 5
    2 [2] LT 0, R0, 1
    3 [2] JMP R0, 6
    4 [3] SUB R0, R0, 1
    5 [3] JMP R0, 7
    6 [5] ADD R0, R0, 1
    7 [6] RETURN R0, 1
  ```

  ```lua
  local a = 5
  if a < 1 then
    a = a - 1
  else
    a = a + 1
  end
  ```
Assembler syntax

• Each function declaration in the assembler listing actually declares a *prototype*; the main function is the main chunk of the script, and in parentheses we have the number of explicit arguments that the function takes (not counting . . .)

• The disassembler embeds literals directly in instructions that can have them as operands, and fills out the necessary literal array

• In the same way, the assembler figures out how many registers the function uses

• Finally, jumps are *absolute* instead of relative, and can be done to symbolic *labels*, the assembler turns both into offsets
Upvalues

• We have to list the upvalues that the closure will have with .upvalue clauses; we give the name of the upvalue, 0 if it comes from an upvalue of the enclosing function, or 1 if it comes from a register, and either the upvalue index in the enclosing function’s closure or the register

```lua
function counter(0):
  loadk r0, 0
  closure r1, anon
  return r1, 2

function anon(0):
  .upvalue n, 1, 0
  getupval r0, 0
  add r0, r0, 1
  setupval 0, r0 ; yes, this is backwards!
  getupval 0, r0
  return r0, 2

local function counter()
  local n = 0
  return function ()
    n = n + 1
    return n
  end
end
```
• Global variables are actually fields in a table usually stored in upvalue 0:

```lua
function main(0):
    upvalue _ENV, 1, 0
    closure r0, hello
    settabup 0, "n", 5
    move r1, r0
    call r1, 1, 1
    return r0, 1

local function hello()
    n = n + 1
    print("hello world", n)
end

n = 5
hello()
```

```lua
function hello(0):
    upvalue _ENV, 0, 0
    gettabup r0, 0, "n"
    add r0, r0, 1
    settabup 0, "n", r0
    gettabup r0, 0, "print"
    loadk r1, "hello world"
    gettabup r2, 0, "n"
    call r0, 3, 1
    return r0, 1
```
What Lua code corresponds to the instructions below, assuming that R0 is the local variable t, R1 the local variable l and R2 the local variable x?

```
NEWTABLE R0 R0 R0
SETTABLE R0 R1 R2
GETTABLE R1 R0 R1

t = {
    [x] = t,
    l = t[x]
}
```