The Virtual Machine of Lua 5.0

Roberto Ierusalimschy, PUC-Rio
WHAT IS LUA?

- Yet another scripting language...
- Conventional syntax:

```lua
function fact (n)
    if n == 0 then
        return 1
    else
        return n * fact(n - 1)
    end
end

function map (a, f)
    local res = {}%
    for i, v in ipairs(a) do%
        res[i] = f(v)
    end%
    return res
end
```
WHAT IS LUA? (CONT.)

- Associative arrays as single data structure
  - first-class values
  - any value allowed as index (not only strings)
  - very efficient implementation
  - syntactic sugar: \texttt{a.x} for \texttt{a["x"]}

- Several not-so-conventional features
  - first-class functions, lexical scoping, proper tail call, coroutines, “dynamic overloading”
Why Lua?

- Light
  - simple and small language, with few concepts
  - core with approximately 60K, complete executable with 140K
- Portable
  - written in “clean C”
  - runs in PalmOS, EPOC (Symbian), Brew (Qualcomm), Playstation II, XBox, embedded systems, mainframes, etc.
- Efficient
  - see benchmarks
- Easy to embed
  - C/C++, Java, Fortran, Ruby, OPL (EPOC), C#
SOME APPLICATIONS

- Games
  - LucasArts, BioWare, Microsoft, Relic Entertainment, Absolute Studios, Monkeystone Games, etc.

- Other Uses
  - tomsrtbt - "The most Linux on one floppy disk"
  - Crazy Ivan Robot (champion of RoboCup 2000/2001 in Denmark)
  - chip layouts (Intel)
  - APT-RPM (Conectiva & United Linux)
  - Space Shuttle Hazardous Gas Detection System (ASRC Aerospace)
Poll from gamedev.net

Which language do you use for scripting in your game engine?

<table>
<thead>
<tr>
<th>Language</th>
<th>Percentage</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>My engine doesn’t have scripting</td>
<td>27.3%</td>
<td>188</td>
</tr>
<tr>
<td>I made my own</td>
<td>26.3%</td>
<td>181</td>
</tr>
<tr>
<td>Lua</td>
<td>20.5%</td>
<td>141</td>
</tr>
<tr>
<td>C (with co-routines)</td>
<td>9.75%</td>
<td>67</td>
</tr>
<tr>
<td>Python</td>
<td>6.98%</td>
<td>48</td>
</tr>
<tr>
<td>Lisp</td>
<td>1.45%</td>
<td>10</td>
</tr>
<tr>
<td>Perl</td>
<td>1.31%</td>
<td>9</td>
</tr>
<tr>
<td>Ruby</td>
<td>1.16%</td>
<td>8</td>
</tr>
<tr>
<td>TCL</td>
<td>0.58%</td>
<td>4</td>
</tr>
<tr>
<td>Other</td>
<td>4.51%</td>
<td>31</td>
</tr>
</tbody>
</table>
Enter door to LUA Bar
Virtual Machine

- Most virtual machines use a stack model
  - heritage from Pascal *p-code*, followed by Java, etc.

- Example in Lua 4.0:

```lua
while a<lim do a=a+1 end

3 GETLOCAL 0 ; a
4 GETLOCAL 1 ; lim
5 JMPGE 4 ; to 10
6 GETLOCAL 0 ; a
7 ADDI 1
8 SETLOCAL 0 ; a
9 JMP -7 ; to 3
```
Another Model for Virtual Machines

- Stack-machine instructions are too low level
- Interpreters add high overhead per instruction
- Register machines allow more powerful instructions
  
  ADD 0 0 [1] ; a=a+1

- Overhead to decode more complex instruction is compensated by fewer instructions
- “registers” for each function are allocated on the execution stack at activation time
  - large number of registers (up to 256) simplifies code generation
INSTRUCTION FORMATS

• Three-argument format, used for most operators
  ◦ binary operators & indexing

<table>
<thead>
<tr>
<th>31</th>
<th>23 22</th>
<th>14 13</th>
<th>6  5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>B</td>
<td>A</td>
<td>OP</td>
<td></td>
</tr>
</tbody>
</table>

• All instructions have a 6-bit opcode
  ◦ the virtual machine in Lua 5.0 uses 35 opcodes

• Operand A refers to a register
  ◦ usually the destination
  ◦ limits the maximum number of registers per function

• Operands B and C can refer to a register or a constant
  ◦ a constant can be any Lua value, stored in an array of constants private to each function
**INSTRUCTION EXAMPLES**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 0</th>
<th>Register 1</th>
<th>Index 2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>259</td>
<td>; a = a+1</td>
</tr>
<tr>
<td>DIV</td>
<td>0</td>
<td>259</td>
<td>0</td>
<td>; a = 1/a</td>
</tr>
<tr>
<td>GETTABLE</td>
<td>0</td>
<td>1</td>
<td>260</td>
<td>; a = t.x</td>
</tr>
<tr>
<td>SETTABLE</td>
<td>0</td>
<td>1</td>
<td>260</td>
<td>; t.x = a</td>
</tr>
</tbody>
</table>

- assuming that the variable a is in register 0, t is in register 1, the number 1 is at index 3 in the array of constants, and the string "x" is at index 4.
There is an alternative format for instructions that do not need three arguments or with arguments that do not fit in 9 bits:
- used for jumps, access to global variables, access to constants with indices greater than 256, etc.
INSTRUCTION EXAMPLES

GETGLOBAL 0 260 ; a = x
SETGLOBAL 1 260 ; x = t
LT 0 259 ; a < 1 ?
JMP * 13

- assuming that the variable a is in register 0, t is in register 1, the number 1 is at index 3 in the array of constants, and the string "x" is at index 4.

- conceptually, LT skips the next instruction (always a jump) if the test fails. In the current implementation, it does the jump if the test succeed.

- jumps interpret the $B_x$ field as a signed offset (in excess-$2^{17}$)
(all variables are local)

```plaintext
while i<lim do a[i] = 0 end
```

**-- Lua 4.0**

```
2 GETLOCAL 2 ; i
3 GETLOCAL 1 ; lim
4 JMPGE 5 ; to 10
5 GETLOCAL 0 ; a
6 GETLOCAL 2 ; i
7 PUSHINT 0
8 SETTABLE
9 JMP -8 ; to 2
```

**-- Lua 5.0**

```
2 JMP * 1 ; to 4
3 SETTABLE 0 2 256 ; a[i] = 0
4 LT * 2 1 ; i < lim?
5 JMP * -3 ; to 3
```
IMPLEMENTATION OF TABLES

- Each table may have two parts, a “hash” part and an “array” part

Example: \( \{ n = 3; 100, 200, 300 \} \)
**TABLES: HASH PART**

- Hashing with internal lists for collision resolution
- Run a *rehash* when table is full:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>val</td>
<td>⬤</td>
</tr>
<tr>
<td>nil</td>
<td></td>
<td>⬤</td>
</tr>
</tbody>
</table>

  → insert key 4 →

- Avoid secondary collisions, moving old elements when inserting new ones

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>val</td>
<td></td>
</tr>
<tr>
<td>nil</td>
<td></td>
<td>⬤</td>
</tr>
<tr>
<td>nil</td>
<td></td>
<td>⬤</td>
</tr>
<tr>
<td>4</td>
<td>val</td>
<td>⬤</td>
</tr>
</tbody>
</table>

  → insert key 3 →
Problem: how to distribute elements among the two parts of a table?
  ○ or: what is the best size for the array?

Sparse arrays may waste lots of space
  ○ A table with a single element at index 10,000 should not have 10,000 elements

How should next table behave when we try to insert index 5?

\[
a = \{n = 3; 100, 200, 300\}; \quad a[5] = 500
\]
COMPUTING THE SIZE OF A TABLE

• When a table rehashes, it recomputes the size of both its parts

• The array part has size $N$, where $N$ satisfies the following rules:
  - $N$ is a power of 2
  - the table contains at least $N/2$ integer keys in the interval $[1, N]$
  - the table has at least one integer key in the interval $[N/2 + 1, N]$

• Algorithm is $O(n)$, where $n$ is the total number of elements in the table
Computing the size of a table (cont.)

- Basic algorithm: to build an array where $a_i$ is the number of integer keys in the interval $(2^{i-1}, 2^i]$
  - array needs only 32 entries

- Easy task, given a fast algorithm to compute $\lfloor \log_2 x \rfloor$
  - the index of the highest one bit in $x$
Now, all we have to do is to traverse the array:

```plaintext
total = 0
bestsize = 0
for i=0,32 do
    if a[i] > 0 then
        total += a[i]
        if total >= 2^(i-1) then
            bestsize = i
        end
    end
end
```
## Performance

<table>
<thead>
<tr>
<th>program</th>
<th>Lua 4.0</th>
<th>Lua 5’</th>
<th>Lua 5.0</th>
<th>Perl 5.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>random (1e6)</td>
<td>1.03s</td>
<td>0.92s</td>
<td>1.08s</td>
<td>1.64s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(89%)</td>
<td>(105%)</td>
<td>(159%)</td>
</tr>
<tr>
<td>sieve (100)</td>
<td>0.94s</td>
<td>0.79s</td>
<td>0.62s</td>
<td>1.29s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(84%)</td>
<td>(66%)</td>
<td>(137%)</td>
</tr>
<tr>
<td>heapsort (5e4)</td>
<td>1.04s</td>
<td>1.00s</td>
<td>0.70s</td>
<td>1.81s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(96%)</td>
<td>(67%)</td>
<td>(174%)</td>
</tr>
<tr>
<td>matrix (50)</td>
<td>0.89s</td>
<td>0.78s</td>
<td>0.58s</td>
<td>1.13s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(87%)</td>
<td>(65%)</td>
<td>(127%)</td>
</tr>
<tr>
<td>fibo (30)</td>
<td>0.74s</td>
<td>0.66s</td>
<td>0.69s</td>
<td>2.91s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(89%)</td>
<td>(93%)</td>
<td>(392%)</td>
</tr>
<tr>
<td>ack (8)</td>
<td>0.91s</td>
<td>0.84s</td>
<td>0.84s</td>
<td>4.77s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(92%)</td>
<td>(92%)</td>
<td>(524%)</td>
</tr>
</tbody>
</table>

- all test code copied from *The Great Computer Language Shootout*
- Lua 5’ is Lua 5.0 without table-array optimization, tail calls, and dynamic stacks (related to coroutines).
- percentages are relative to Lua 4.0.
**Final Remarks**

- Compiler for register-based machine is more complex
  - needs some primitive optimizations to use registers

- Interpreter for register-based machine is more complex
  - needs to decode instructions

- Requirements
  - no more than 256 local variables and temporaries

- Main gains:
  - avoid moves of local variables and constants
  - fewer instructions per task
  - potential gain with CSE optimizations