The Virtual Machine of Lua 5.0

Roberto Ierusalimschy, PUC-Rio

WHAT IS LUA?

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

```
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: `a.x` for `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?

My engine doesn’t have scripting	27.3%	188
I made my own	26.3%	181
Lua	20.5%	141
C (with co-routines)	9.75%	67
Python	6.98%	48
Lisp	1.45%	10
Perl	1.31%	9
Ruby	1.16%	8
TCL	0.58%	4
Other	4.51%	31

Enter door to LUA Bar

05.07.01

VIRTUAL MACHINE

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

- Example in Lua 4.0:

```
  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

31 23	22 14	13 6	5 0
`C`	`B`	`A`	`OP`

- 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

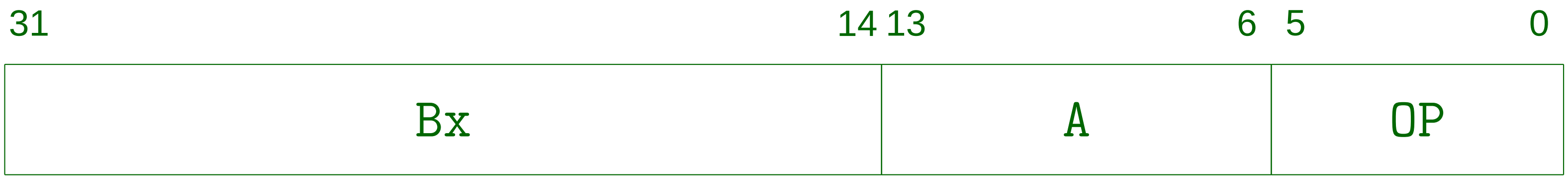
INSTRUCTION EXAMPLES

```
ADD        0  0    259      ; a = a+1

DIV        0  259  0        ; a = 1/a

GETTABLE   0  1    260      ; a = t.x

SETTABLE   0  1    260      ; t.x = a
```

- 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.

INSTRUCTION FORMATS

- 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.



31 … 14	13 … 6	5 … 0
Bx	A	OP

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 `Bx` field as a signed offset (in excess-2^{17})

CODE EXAMPLE

(all variables are local)

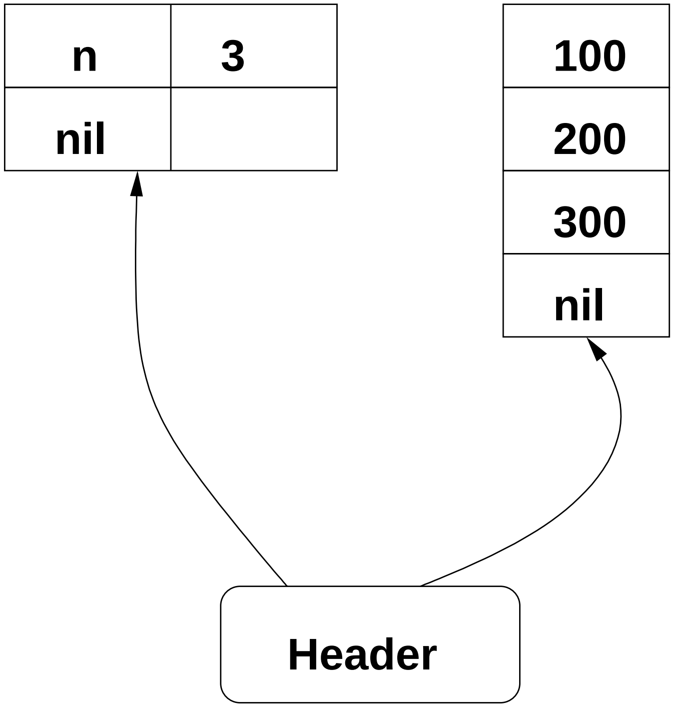
```
while i<lim do a[i] = 0 end
```

```
 -- 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
```

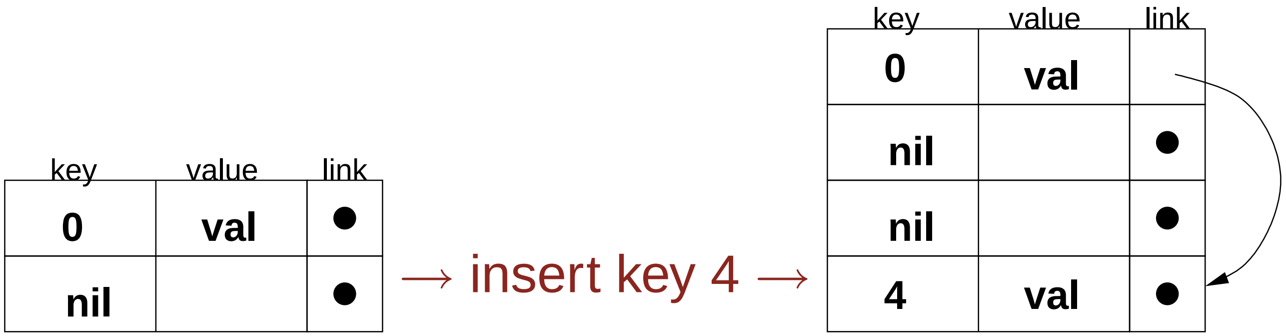
IMPLEMENTATION OF TABLES

- 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:

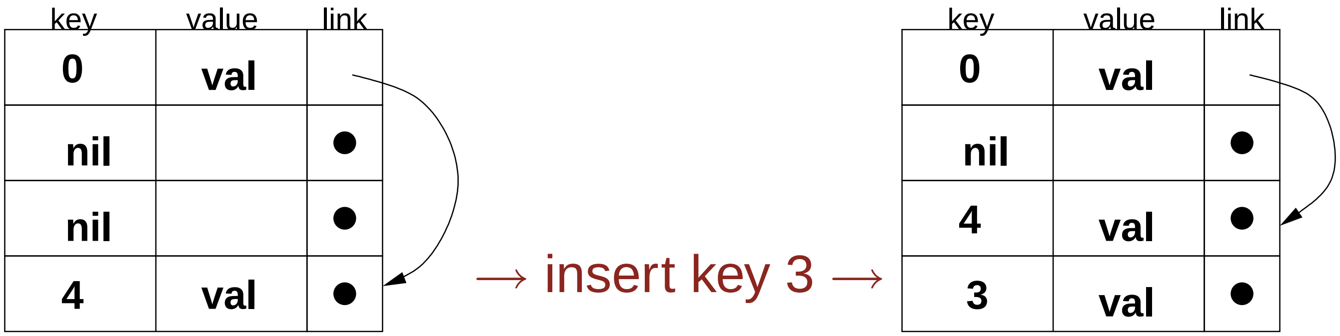


key	value	link
0	**val**	●
nil		●

→ insert key 4 →

key	value	link
0	**val**	
nil		●
nil		●
4	**val**	●

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



key	value	link
0	**val**	
nil		●
nil		●
4	**val**	●

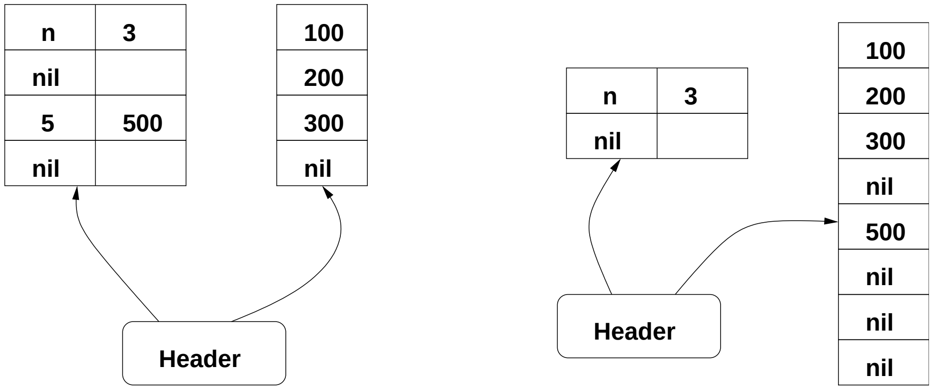
→ insert key 3 →

key	value	link
0	**val**	
nil		●
4	**val**	●
3	**val**	●

TABLES: ARRAY PART

- 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}; 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 $\mathrm{O}(n)$, 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

COMPUTING THE SIZE OF A TABLE (CONT.)

- Now, all we have to do is to traverse the array:

```
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

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

- 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