

The Implementation of Lua 5.0

Roberto Ierusalimschy
Luiz Henrique de Figueiredo
Waldemar Celes



MAIN GOALS

- Portability
 - ANSI C and C++
 - avoid dark corners
- Simplicity
 - small size
- Efficiency

VALUES AND OBJECTS

- Values represent all Lua values
- Objects represent values that involve memory allocation
 - strings, tables, functions, heavy userdata, threads
- Representation of Values: tagged unions

```
typedef union {  
    GCObject *gc;  
    void *p;  
    lua_Number n;  
    int b;  
} Value;
```

```
typedef struct lua_TValue {  
    Value value;  
    int tt  
} TValue;
```

OBJECTS

- Pointed by field `GCObject *gc` in values
- Union with common head:

```
GCObject *next; lu_byte tt; lu_byte marked
```

- Redundant tag used by GC
- Strings are hibrid
 - Objects from an implementation point of view
 - Values from a semantics point of view

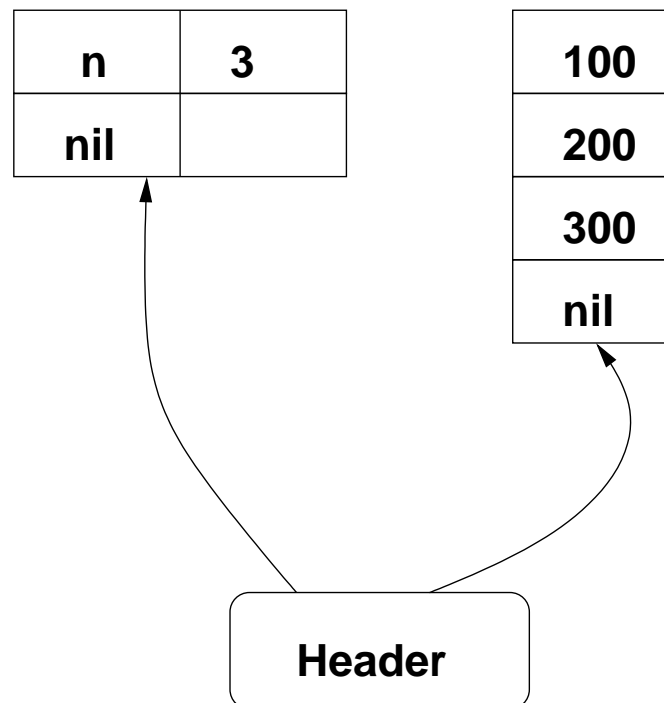
STRINGS

- Represented with explicit length
- *Internalized*
 - save space
 - save time for comparison/hashing
 - more expensive when creating strings

IMPLEMENTATION OF TABLES

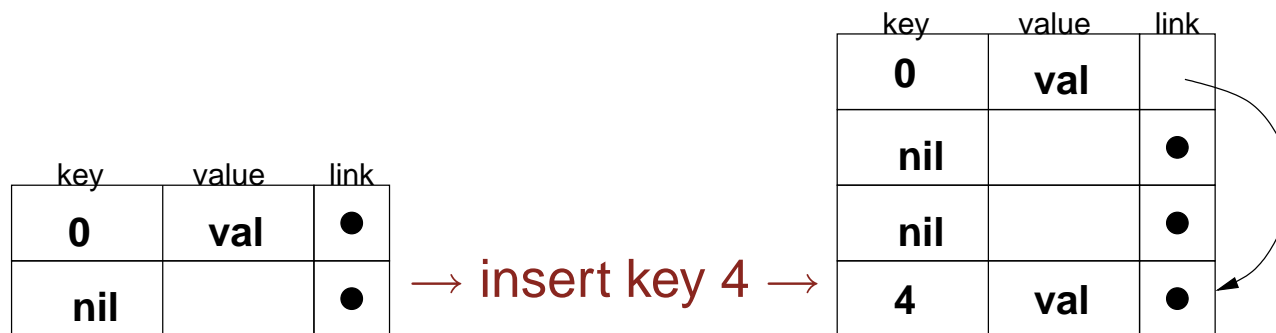
- Each table may have two parts, a “hash” part and an “array” part

- Example: `{n = 3; 100, 200, 300}`



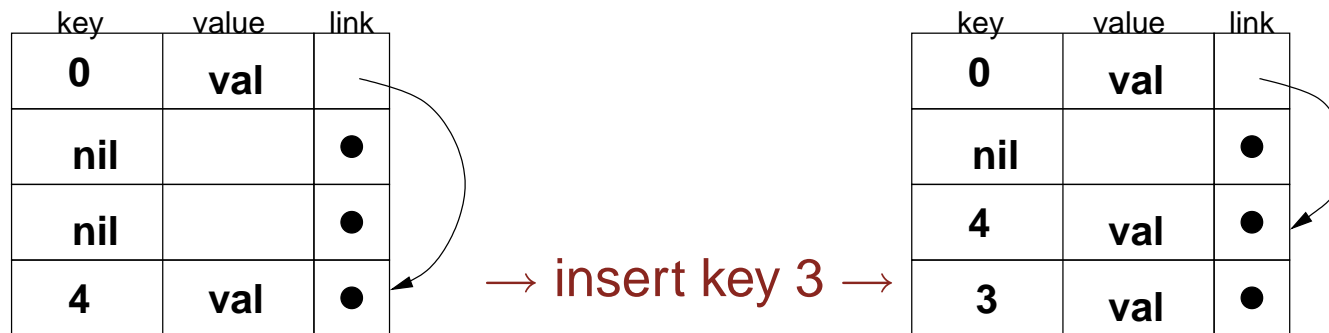
TABLES: HASH PART

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



TABLES: HASH PART (2)

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



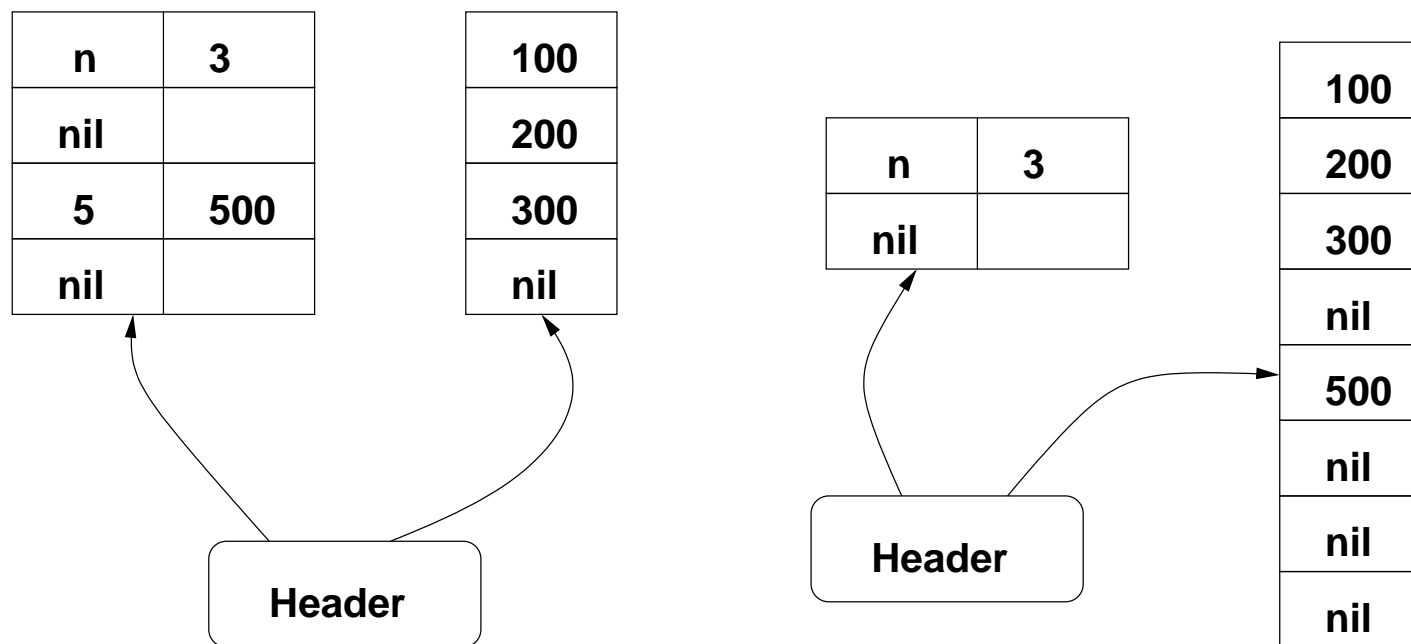
TABLES: ARRAY PART

- 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

TABLES: ARRAY PART (2)

- 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 $O(n)$, where n is the total number of elements in the table

COMPUTING THE SIZE OF A TABLE (2)

- 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 (3)

- 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
end
```

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



- All instructions have a 6-bit opcode
- Operand A refers to a register
- 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.



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 succeeds.
- 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
```

CLOSURES

- Lua has first-class functions with lexical scoping
- Variables in each function may have different scopes

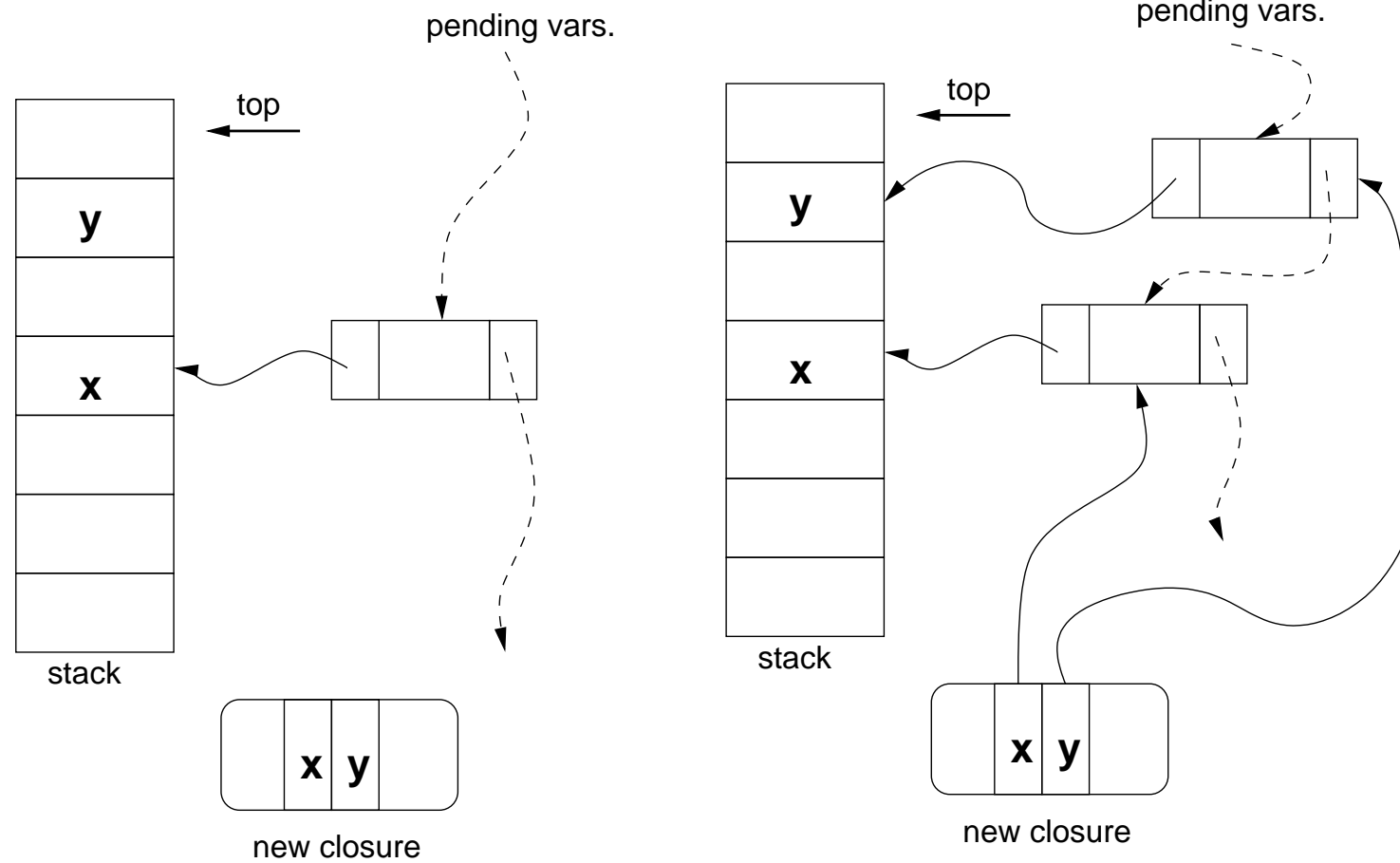
```
local a =  
local x = 0  
for i = 1, N do  
  local y = i  
  a[i] = function (z) y=z; return x+y end  
end
```

- Small implementation rules out complex algorithms
- One-pass compiler cannot know in advance whether a variable is used by a inner function
 - How to assign variables to the stack?

IMPLEMENTATION OF CLOSURES (1/4)

- All variables go to the stack
- Use of *upvalues* to represent external variables
- When a Closure is created, it searches for an upvalue for each of its external variables
 - if upvalue not found, create a new one
 - search is fast, because lists are typically very short

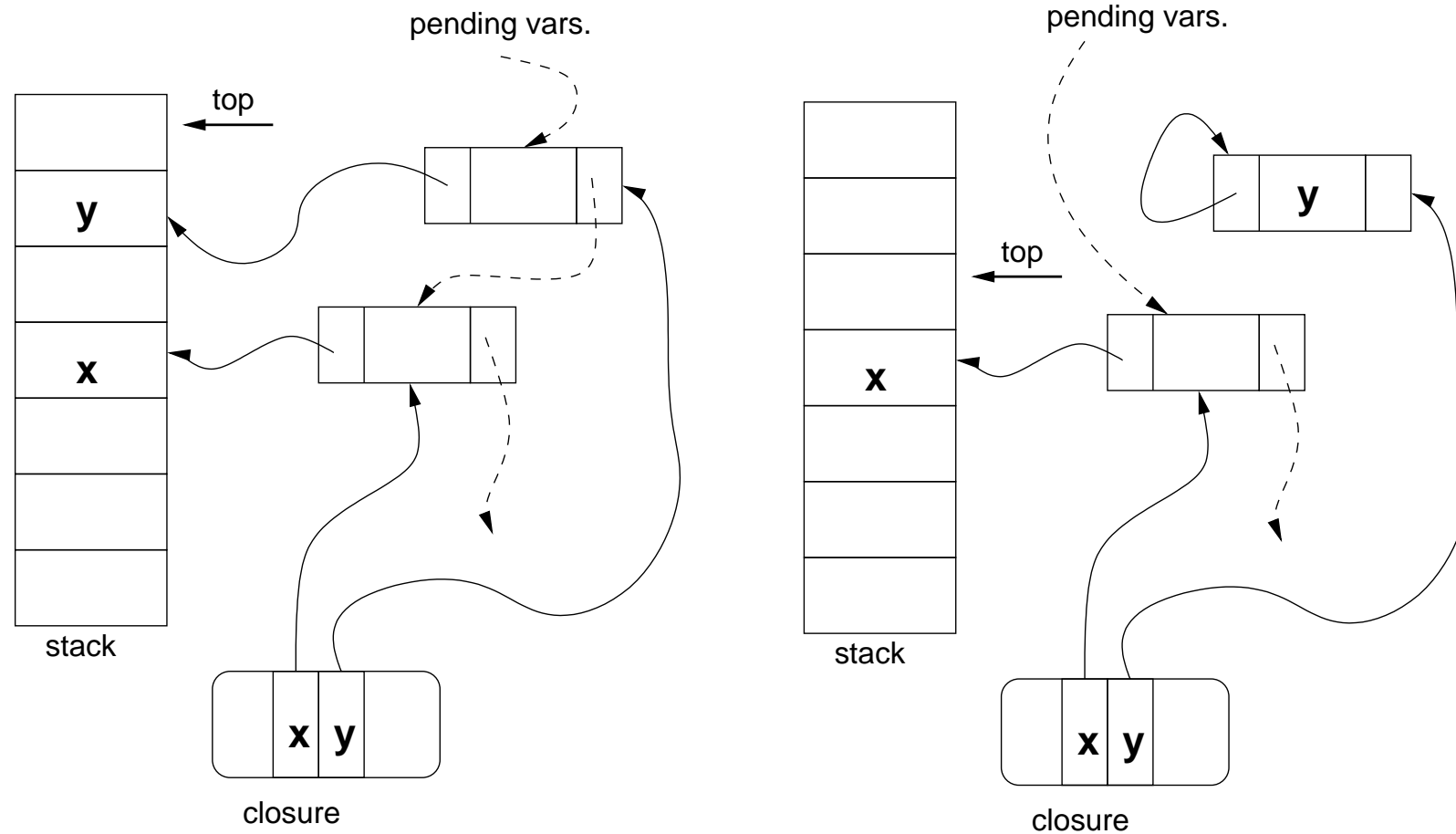
IMPLEMENTATION OF CLOSURES (2/4)



IMPLEMENTATION OF CLOSURES (3/4)

- All accesses to external variables are through a pointer in the upvalue
- When an upvalue in the stack goes out of scope, the upvalue is *closed*
 - upvalue is removed from list
 - the value is copied to an area in the upvalue itself
 - the upvalue points to itself

IMPLEMENTATION OF CLOSURES (4/4)



INCREMENTAL GARBAGE COLLECTOR

- Uses a three-color algorithm
 - white objects are not marked
 - grey objects are marked but not traversed
 - black objects are marked and traversed
- Main invariant: black objects never point to white objects
- Well known, but with several undocumented details

THREE-COLOR ALGORITHM: MARK

- Mark root objects as grey
- At each step, traverses a grey object (turning it black) and mark new accessible objects as grey
- Stops when there are no more grey objects; white objects are garbage
- Write barrier detects when storing a white object into a black one
- Optimization: stacks are always grey
 - avoids barrier when writing to stacks

THREE-COLOR ALGORITHM: ATOMIC STEP

- Traverses stacks
- Separates dead userdata with finalizers
- Traverses them
- Clears weak tables
- Changes white objects to dead
 - trick: toggles between two whites

THREE-COLOR ALGORITHM: SWEEP

- Sweep all objects
- Collect dead objects
- Change black objects to white

GARBAGE-COLLECTOR DETAILS

- Upvalues x dead threads
 - dead threads are not traversed in the atomic step but when alive they may have changed value pointed by an upvalue
 - because threads have no barrier, upvalue may point to dead object
- solution: traverse all open upvalues in the atomic step

GARBAGE-COLLECTOR DETAILS (2)

- Granularity
 - several tasks are atomic
 - seems to be no problem in real use
- Step size
 - how much to do at each step?
 - how to compare ?step size? across different phases?
- Collector speed
 - stops between steps and between collections

FINAL REMARKS

- Lexical scoping: no overhead for non users
- Virtual machine: good performance gains
 - plus potential gains with CSE optimizations
 - compiler for register-based machine is more complex
- Representation for tables
 - arrays save more than 50% memory
 - efficient representation both for dense and sparse arrays
- Incremental Garbage Collector