LuaDec – a Lua decompiler

Hisham Muhammad
<hisham@inf.puc-rio.br>

Lua Workshop 2005

LuaDec

- Programming assignment, 2004
  - Good way to learn about the Lua VM
- Targets Lua 5.0.2
- Written in C
- Based on the Luac disassembler

Writing a decompiler for Lua

- High-level opcodes
- Lots of symbolic information
- Registers map to local variables
- No goto
- Single compiler to target
- Not as easy as with stack machines (Java)

Rebuilding constructs

- Decompiler performs two passes
- First pass:
  - Identify jumps
  - Mark position of scope blocks closed by the CLOSE opcode
- Second, main pass:
  - Symbolic interpretation
  - Recursively process functions, following the CLOSURE opcode

First pass

- A JMP opcode means we need to emit some code on the other end of the construct
- A backward JMP to an instruction after a forward JMP is a “while”
- To an instruction after a TFORPREP is a “for”
- Otherwise, is a “repeat” block

Symbolic interpretation

- Run through the code keeping track of registers

```
x[a+b]=y[c+d]     0   1   2   3   4   5   6    7
                  a   b   c   d   x   y
ADD         6 0 1   a   b   c   d   x   y   a+b
ADD         7 2 3   a   b   c   d   x   y   a+b  c+d
GETTABLE 7 5 7   a   b   c   d   x   y   a+b  y[c+d]
SETTABLE 4 6 7   a   b   c   d   x   y   a+b  y[c+d]
```

Locals allocate registers

```
local a,b,c
a = 1
b = 2
c = a + b
local d = 4
c = a + d
b = 10
c = a + b
```

```
1 LOADNIL 0 2
2 LOADK   0 0
3 LOADK   1 1
4 ADD     2 0 1
5 LOADK   3 2
6 ADD     2 0 3
7 LOADK   1 3
8 ADD     2 0 1
```

constants

0	1
1	2
2	4
3	10

locals

0	a	1-8
1	b	1-8
2	c	1-8
3	d	5-8

Locals allocate registers

```
local a,b,c
a = 1
b = 2
c = a + b
local d = 4
c = a + d
b = 10
c = a + b
```

```
1 LOADNIL a c
2 LOADK   a 1
3 LOADK   b 2
4 ADD     c a b
5 LOADK   d 4
6 ADD     c a d
7 LOADK   b 10
8 ADD     c a b
```

When to output code

- As late as possible
- We have enough information about the locals
  - No need to add temporary variables
- As assignments happen, keep a list of “pending variables”
- Only output a pending variable when it is overwritten (or at the end of the block)

When to output code

- Treat “variable registers” and “temporary registers” differently
- Necessary for correctness

```
a, b = b, a      0   1   2
                 a   b
MOVE   2 1       a   b   b
MOVE   1 0       a   a   b
MOVE   0 2       b   a   b
```

Boolean conditions

- Turning a series of calculations, tests and jumps into an expression, taking into account:
  - Short circuit
  - Nested if's
  - Relational constructs in assignments

Building an expression

- As expressions resulting in pairs of relational tests and jumps are read, they are collected in a list
- Translation into a boolean expression:
  - Identify jumps to “then” and “else” addresses
  - Devise parenthesis scheme, build a tree
  - “Print” expression, based on context (conditions may be inverted)

```
 1 LOADNIL 0 2 0                 local a, x, y
 2 JMP     0 16    ; to 19       while x do
 3 EQ      0 1 250 ; - 2           if ((x==2) and y)
 4 JMP     0 2     ; to 7            or ((x==3) and 1) or 0
 5 TEST    2 2 1                   then
 6 JMP     0 5     ; to 12             a = 1
 7 EQ      0 1 251 ; - 3               do break end
 8 JMP     0 3     ; to 12             a = 2
 9 LOADK   3 2     ; 1               else
10 TEST    3 3 1                       a = 3
11 JMP     0 0     ; to 12             do break end
12 LOADK   0 2     ; 1                 a = 4
13 JMP     0 7     ; to 21           end
14 LOADK   0 0     ; 2               a = 5
15 JMP     0 3     ; to 19       end
16 LOADK   0 1     ; 3
17 JMP     0 3     ; to 21
18 LOADK   0 4     ; 4
19 TEST    1 1 1
20 JMP     0 -18   ; to 3
21 RETURN  0 1 0
```

Status

- Still gets confused with complex expressions
  - Fundamental limitation: no block analysis
- Successfully decompiles all demos in the test/ directory
- After a few revisions, it now survives a good deal of Roberto's stress tests

Avoiding decompilation

- LuaDec relies on the locals table
  - luac -s confuses it
- It's easy to obfuscate your bytecode
  - For example, swap opcodes around
- Reading Lua VM code is easy for a human
  - If you have any secrets, use encryption

Conclusions

- A decompiler for a high-level register machine
  - Impossible to make a perfect decompiler for arbitrary bytecode
- Opportunities for optimizations in Lua bytecode
  - Offline compiler
- Not actively maintained (any takers?)