PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Testing Lua

Roberto Ierusalimschy

```
#define modff(x,fp)  \
   ((float)modf((double)(x), (double *)(fp)))
```

```
#define sprintf(s,fmt,...)  \
    snprintf(s, sizeof(s), fmt, __VA_ARGS__)
```

```
  if ((err = SSLHashSHA1.update(...)) != 0)
    goto fail;
    goto fail;
  [...]
  err = sslRawVerify(...);
fail:
  return err;
```

Testing Lua: Goals

- Improve the quality (robustness) of our code
- Expose bugs
- Crash the interpreter
  - earn extra points!

Testing Lua: No Goals

- Help debugging
  - Not structured in “increasing complexity” order
  - No unit testing
  - Too much interdependencies
- Test conformance for alternative implementations
  - Too much dependency on “implementation details”
    - error messages
    - GC behavior
    - Internal tests

Before Testing

- Warnings:

```
-pedantic
-Wextra
-Wshadow
-Wsign-compare
-Wundef
-Wwrite-strings
-Wredundant-decls
-Wdisabled-optimization
-Wdeclaration-after-statement
-Wmissing-prototypes
-Wnested-externs
-Wstrict-prototypes
-Wc++-compat
-Wold-style-definition
```

Assertions

- Heavy use of macros in the code to insert assertions
- Expression checks

```
#if defined(lua_assert)
#define check_exp(c,e)   (lua_assert(c), (e))
#else
#define check_exp(c,e)   (e)
#endif
```

Assertions

- All accesses to objects check their liveness
- All accesses to union fields check corresponding tag
- Functions in the C API check preconditions
  - what can be practically checked in C
  - can be turned on separately, in production code

```
/* test for sizes in 'l_sprintf' */
#undef l_sprintf
#define l_sprintf(s,sz,f,i) \
  (memset(s,0xAB,sz), snprintf(s,sz,f,i))
```

The Bad News

- Portability
- C
- libc
- Inherent complexity of a language/interpreter
  - any input is valid

Portability

- Numeric types overflow differently in different machines
- Some conditions are impossible in some architectures
  - overflows of `size_t` in 64-bit machines
  - different integer types may not lose precision in some architectures
- Different libc can give different results for some operations
  - format `'%a'`

C

- It is almost impossible to write correct C code according to the letter of the standard

*“The accuracy of the floating-point operations (+, -, *, /) and of the library functions in <math.h> and <complex.h> that return floating-point results is implementation-defined. The implementation may state that the accuracy is unknown.”*

```
assert(2^3 == 8)
```

The Good News

- Small code base
  - ~23 000 lines of code
  - less than 10 000 LLOCs (semicolons)
- Lua is quite flexible for writing (its own) tests
- Tests do not need (much) performance

The Tests

- ~12 000 lines of Lua code in ~30 files
  - ~half the size of Lua
  - Plus 1 700 lines of C (test library)
- One main file `'all.lua'` runs all the others
  - ~40 sec. in my “old Pentium machine”
  - ~5 sec. In my “new Intel machine”
- Mostly automated

The Tests

- Coverage: 99.3% of the interpreter, API, and standard libraries
  - 53 lines (out of 8148) not covered
    - mostly overflow tests and other hard-to-create errors
- Some global variables allow skipping some parts:
  - `_soft`: avoids resource-intensive tests
  - `_port`: avoids non-portable tests (e.g., those that need a Posix shell)

Some Guidelines

- “A feature does not exist unless there is a test for it.” (*Extreme Programming*)
- Tests should be tested, too!
- Each bug reported in Lua generates (at least) one new test that must catch that bug
- Always remember your goal!
  - find bugs
  - crash Lua

Manual Parts

- Multiple platforms
  - mainly Linux (Ubuntu) 64 bits and Windows 7 32 bits
- Multiple configurations
  - different numeric types
  - compatibility options

Manual Parts

- A few hard tests also need special compiler options or separate invocation
  - stack reallocation
  - emergency garbage collection
  - memory traversal
- Testing the tests
  - remove/change details in code

The Basics

- Lots of very conventional (and boring) stuff

```
-- testing string.sub
assert(string.sub("123456789",2,4) == "234")
assert(string.sub("123456789",7) == "789")
assert(string.sub("123456789",7,6) == "")
assert(string.sub("123456789",7,7) == "7")
assert(string.sub("123456789",0,0) == "")
assert(string.sub("123456789",-10,10) == "123456789")
```

Errors

- `pcall` (p*rotected call*) is a key ingredient for testing errors and error messages

```
local function check (msg, f, ...)
  local s, err = pcall(f, ...)
  assert(not s and string.find(err, msg))
end

check("out of limits", pack, "i0", 0)
check("out of limits", pack, "i17", 0)
check("out of limits", pack, "!17", 0)
check("invalid format option 'r'", pack, "i3r", 0)
check("not power of 2", pack, "!4i3", 0);
```

Dynamic Code Generation

- More extreme test cases can be generated by Lua

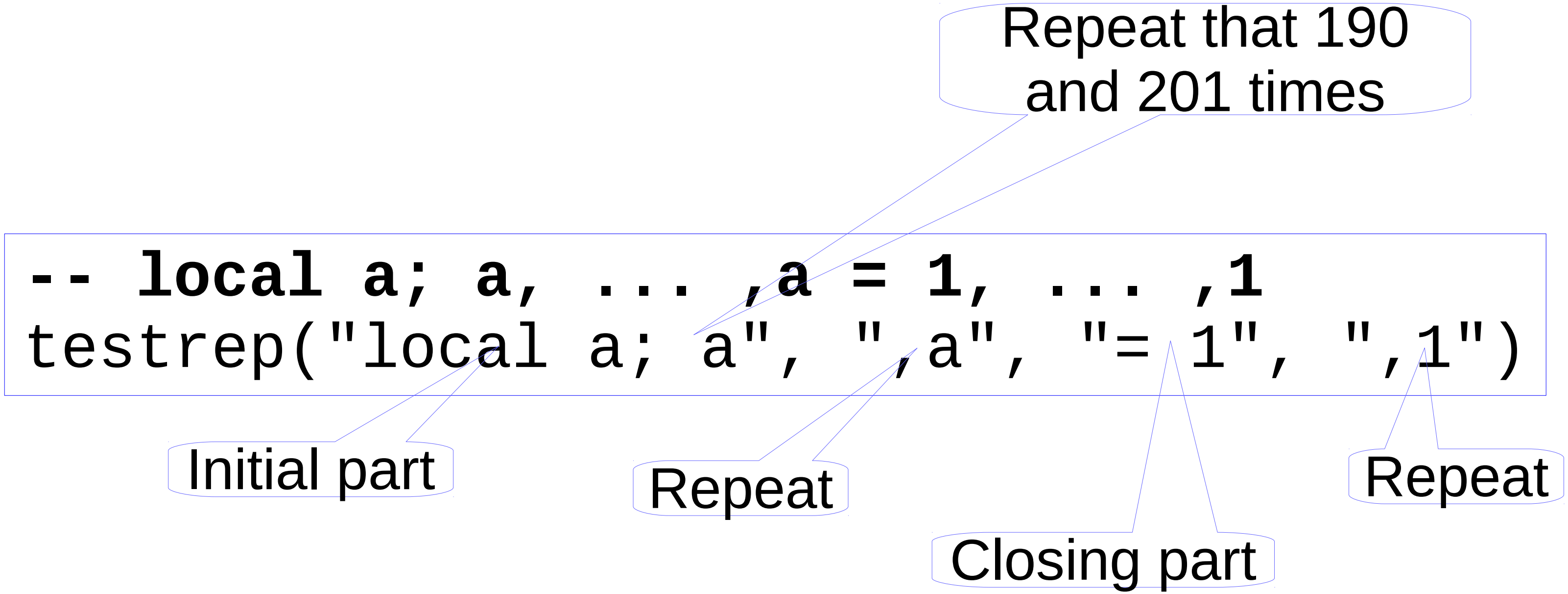
```
-- 0xffff...fff.0
assert(tonumber("0x" .. string.rep("f", 150) ..
                ".0") == 2.0^(4*150) - 1)

-- 0x.000 ... 0074p4004
assert(tonumber('0x.' .. string.rep('0', 1000) ..
                '74p4004') == 0x7.4)
```

- Some of them can be really extreme...

```
-- chunk with infinite lines

local manynl = string.rep("\n", 1e6)

local function nl ()
  return manynl
end

local st, msg = load(nl)

assert(not st and string.find(msg, "too many lines"))
```

Syntax Limits

- Test large syntactical structures both inside and outside limits



```
-- local a; a, ... ,a = 1, ... ,1
testrep("local a; a", ",a", "= 1", ",1")
```

```
-- local a; a, ... ,a = 1, ... ,1
testrep("local a; a", ",a", "= 1", ",1")

-- local a; a = {...{0}...}
testrep("local a; a=", "{", "0", "}")

-- local a; a = (...(2)...)

-- local a; a(a(a(...a(2)...)))

-- do ... do end ... end

-- while a do while a do ... end end

-- a = a..a..a..a.. ... ..a
```

```
local function testrep (init, rep, close, repc)

  local s = init ..
            string.rep(rep, maxClevel - 10) ..
            close ..
            string.rep(repc, maxClevel - 10)

  assert(load(s))   -- 190 levels is OK

  s = init .. string.rep(rep, maxClevel + 1)

  checkmessage(s, "too many C levels")

end
```

Dynamic Code Generation

- To test short-circuit optimizations, try all combinations of some operands connected with `and's` and `or's`

```
[...]
(nil and (nil and nil))
(false or (nil and 10))
[...]
((nil or true) or 10)
[...]
(((10 or true) and true) and false)
(((10 or true) and true) or false)
(((10 or true) and true) and true)
(((10 or true) and true) or true)
(((10 or true) and true) and 10)
```

Dynamic Code Generation

- With their respective correct values...

```
local basiccases = { {"nil", nil}, {"false", false},
  {"true", true}, {"10", 10},
  {"(_ENV.GLOB1 < _ENV.GLOB2)", true},
  {"(_ENV.GLOB2 < _ENV.GLOB1)", false},
}

local binops = {
  {" and ", function (a,b)
     if not a then return a else return b end end},
  {" or ", function (a,b)
     if a then return a else return b end end},
}
```

```
local mem = {basiccases}    -- for memoization

local function allcases (n)
  if mem[n] then return mem[n] end
  local res = {}
  for i = 1, n - 1 do
    for _, v1 in ipairs(allcases(i)) do
      for _, v2 in ipairs(allcases(n - i)) do
        for _, op in ipairs(binops) do
            res[#res + 1] = {
              "(" .. v1[1] .. op[1] .. v2[1] .. ")",
              op[2](v1[2], v2[2])
            }
        end
      end
    end
  end
  mem[n] = res   -- memoize
  return res
end
```

The Stand-alone Interpreter

- Most tests for the `lua` application need support from the shell
  - intput/output redirection
  - `stderr` redirection
  - environment variables
  - some extra facilities
- These tests can be skipped defining global variable `_port`

The Stand-alone Interpreter

```
local out = os.tmpname()

-- standard input
RUN('echo "print(10)\nprint(2)\n" | lua > %s', out)
checkout("10\n2\n")

-- test option '-'
RUN('echo "print(arg[1])" | lua - -h > %s', out)
checkout("-h\n")
```

The Stand-alone Interpreter

```
-- test errors in LUA_INIT
NoRun('LUA_INIT:1: 10',   -- expected error message
      'env LUA_INIT="error(10)" lua')

-- errors in Lua options
NoRun("unrecognized option '-Ex'", "lua -Ex")
```

Ctrl-C

```
local script = [[
  pcall(function () print(12); while true do end end)
  print(42)
]]

local shsc = ('lua -e "%s" & echo $!'):format(script)

local f = io.popen(shsc, "r")
local pid = f:read()
assert(f:read() == "12")
assert(os.execute("kill -INT " .. pid))
assert(f:read() == "42")
assert(f:close())
```

The Test Library

- The Test Library (T) is an optional library for Lua that offers several facilities to improve tests
  - disassembler
  - memory allocation
  - garbage collector
  - C API

Disassembler

```
function check (f, ...)
  local arg = {...}
  local c = T.listcode(f)
  for i=1, #arg do
    assert(string.find(c[i], arg[i]))
  end
end

-- infinite loops
check(function () while 1 do local a = 1 end end,
'LOADK', 'JMP', 'RETURN')

check(function () repeat local x = 1 until true end,
'LOADK', 'RETURN')
```

Constant Folding

```
local function checkK (func, val)
  check(func, 'LOADK', 'RETURN')
  local k = T.listk(func)
  assert(#k == 1 and k[1] == val and
         math.type(k[1]) == math.type(val))
  assert(func() == val)
end

checkK(function () return 3^-1 end, 1/3)

checkK(function () return (1 + 1)^(50 + 50) end,
       2^100)
...
```

Custom Memory-Allocation

- Checks size on deallocations
- Checks write violations around memory blocks
- Checks that all memory is free at exit
  - with `atexit`
- Ensures that all calls to `realloc` change the block address
- Can set a limit for total memory in use
  - allocation fails after limit

Memory-Allocation Errors

- Try to force a memory-allocation error at all points allong the path of some specific tasks
  - state creation, coroutine creation, `loadstring`, `dofile`, constructors, file creation, etc.
- Set memory limit 7 bytes more than current use and try to run given task; repeat until it succeeds
- More important part at exit, checking that there were no leaks

“not enough memory”

```
function testamem (s, f)
  collectgarbage(); collectgarbage()
  local M = T.totalmem()
  while true do
    M = M + 7
    T.totalmem(M)   -- set memory limit
    local a, b = pcall(f)
    T.totalmem(0)     -- remove limit
    if a then break end
    collectgarbage()
    end
  end
end
```

Memory Traversal

- The Test Library defines a function `checkmemory` that traverses all internal data structures checking consistency, specifically about garbage collection
- When available, this function is called regularly during the tests
- Sometimes, we add a call to this function for every single step of the garbage collector

Testing the C API

- Several parts of the API are naturally tested by the libraries
- Other parts need extra work
- How to avoid writing this extra work in C?

The Lua-API Mini-language

- The Test Library exports an interpreter for a mini-language of API commands

```
a, b, c = T.testC([[
  pushnum 1;
  pushnum 2;
  pushnum 3;
 return 2
]])

assert(a == 2 and b == 3 and c == nil)
```

The Lua-API Mini-language

- This interpreter has natural access to the stack of its enclosing function

```
t = pack(T.testC([[
  rotate -2 1;
  return *]], 10, 20, 30, 40))

tcheck(t, {10, 20, 40, 30})

t = pack(T.testC([[
  rotate -2 -1;
  Return *]], 10, 20, 30, 40))

tcheck(t, {10, 20, 40, 30})
```

The Lua-API Mini-language

- Using C closures, the library can also create C functions with custom code

```
f = T.makeCfunc([[
  pushnum 1;
  pushnum 2;
  pushnum 3;
  return 2
]])

a, b, c = f()
assert(a == 2 and b == 3 and c == nil)
```

```
-- testing coroutines with C bodies
f = T.makeCfunc([[
        pushnum 102
        yieldk  1 U2
        cannot be here!
]],
[[      # continuation
        pushvalue U3
        pushvalue U4
        gettop
        return .
]], 23, "huu")

x = coroutine.wrap(f)
assert(x() == 102)
eqtab({x()}, {23, "huu"})
```

Testing the Panic Function

- `T.checkpanic(c1, c2)` runs `c1` in a fresh Lua state, with a panic function that runs `c2` and long-jump back to `checkpanic`, which returns the top of the stack (after closing that Lua state)

```
-- trivial error
assert(T.checkpanic("pushstring hi; error") == "hi")
```

Testing the Panic Function

```
-- using the stack inside panic
assert(T.checkpanic("pushstring hi; error;",
        [[checkstack 5 XX
          pushstring ' alo'
          pushstring ' mundo'
          concat 3]]) == "hi alo mundo")

-- memory error
T.totalmem(T.totalmem() + 5000)
assert(T.checkpanic("newuserdata 10000") ==
         "not enough memory")
T.totalmem(0)
```

Special Tests

- “Hard stack test”
  - forces a reallocation of the stack each time there is a test for stack overflow
- “Hard memory tests”
  - forces a complete emergency collection at each memory allocation
- *`-fsanitize=undefined`*, *`-ftrapv`*
- *Valgrind*

Final Remarks

- Tests are essential to avoid stupid mistakes
- A good test suite is an essential tool for maintenance and evolution
- Thinking about the tests improve our understanding of the code
- Testability is an important cost measure for new features

Final Remarks

- There are still lots of bugs to be found
  - ~9 per year since 2003
  - most of them quite well hidden
  - evolution creates new bugs
- Many bugs found in C compilers and libraries, too
  - `2^3`, `tonumber(“.”)`, `fmod(1,inf)`, unsigned → float, indirect tail call, etc.