The Novelties of Lua 5.1

Roberto Ierusalimschy

Parser Reentrant

- Lua can be freely called while parsing a chunk
- New function `load`
- Opens the door for Macro-processing

New Syntax for Long Strings

- `[==[...]==]`
- Also valid for long comments
  - `--[=[...]=]`
- Allows insertion of *any* literal string
  - does not need to end with newline
- Requirements:
  - variable delimiter
  - clear border around delimiter (e.g., `[[[...]]]` does not work)
  - Old `[[...]]` as a special case

New Syntax for Long Strings (2)

- No more nesting
  - string ends with a fix mark
  - simpler description (and implementation)

```
string.find(s, "%[(=*)%[.-]%1]")
```

Coroutine Debug

- Debug library works on any coroutine:

```
print(debug.traceback(co))
```

- On error, coroutines do not unwind the stack
  - can be inspected later

```
ok = coroutine.resume(co)
if not ok then
  print(debug.traceback(co))
end
```

New Mod Operator

- Why Lua did not have it?
  - probably we forgot it :)
- Several uses
  - helps with bitwise operations

New Mod Operator (2)

- Main rule: $a = (a \textbf{ div } b)b + a\%b$
- But $a \textbf{ div } b$ has several possible meanings
  - floor(a/b), ceil(a/b), round(a/b), trunc(a/b)
- Which is best?
- floor has some nice properties
  - $a = b \bmod c$ iff $a\%c = b\%c$
  - $a\%b$ always in range [0..b) for positive b

New Length Operator

- Final syntax: `#t`
- Results in the *length* (or size, or last index) of an array (or list, or sequence)
- Computed in (log *n*) time
  - with very low multiplier
  - faster than `table.getn` even for huge arrays
- No more `table.setn`

New Length Operator (2)

- Subtle (and mostly useless) semantics for lists with holes
  - use explicit size in those cases
- Nice idioms for list manipulation:

```
t[#t+1] = v     -- insertion
print(t[#t])    -- last element
t[#t] = nil     -- removing
```

String Library

- `string.find` split in two functions
  - `string.find` finds patterns
  - `string.match` extracts subpatterns (captures)
- For coherence, `string.gfind` should be renamed `string.gmatch`

Specialized API Functions

- `lua_tointeger/lua_pushinteger`
- `lua_getfield/lua_setfield`
- Frequent cases
- Allows for small optimizations
  - bigger ones for `lua_tointeger`
- `lua_createtable(asize, rsize)`
  - bigger optimizations in specific cases
  - in Lua, constructors do the job

Configurable Memory Allocation

- `lua_newstate` gets as argument an *allocation function*
- Allocation function must work as a generalized `resize`
- Access to original block size
  - memory system does not need to keep it
- Access to an uninterpreted `void *`
  - allow independent states to use different pools

Config. Memory Allocation (2)

- Lua core does not directly access OS services
  - I/O, memory, etc.
  - uses externally-provided functions for that
- Easy to convert the core to a freestanding C environment

New Vararg Mechanism

- ... as new vararg expression

```
function foo (...)
  print(...)
end
```

- Avoids creating excessive tables
- Avoids arbitrary name
- Main chunks are vararg functions

Environments

- C functions and userdata also have environments
  - all *objects* except tables have an environment
- Concept more uniform
- C functions have direct access to their environment
  - pseudo-index
- Userdata environment only for pogrammer's use

Environments (2)

- C-function environments help libraries share common data
- Userdata environments help link between userdata and corresponding Lua objects
  - eaiser than references
  - no problems with cycles

Incremental Garbage Collector

- Main motivation for Lua 5.1
- Uses a three-color algorithm
  - well known, but with several undocumented details
  - main invariant: black objects never point to white objects

Garbage-Collector (2)

- Granularity
  - several atomic tasks
  - 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

New Module System

New Module System

- Not as much change as it seems
- Mostly policies (bad)
- But suggested, not enforced (good)
- Main changes:
  - *require* directly handles C libraries
  - submodules
  - new function *module* facilitates modules to follow suggested policies
  - luaL_openlib does the same for C libraries

require

- First search for a *loader* for the given module
- “preload” table, Lua files, C libraries, “whole-package” C libraries
  - “all-in-one” Lua and or C libraries?
- After finding a loader, calls it with the module name

Whole-Package C Libraries

- Given module `a.b.c`, search for C file `a`
- If found, look for function `luaopen_a_b_c` to load module
- Same DLL may provide open functions for different modules
- Do we need an “all-in-one” loader?

“Ignore Mark”

- When building `luaopen_` name, `require` ignores everything before a “`:`”
  - `:mod` ⇨ `luaopen_mod`
  - `v1_3:mod` ⇨ `luaopen_mod`
  - `a.b.:c` ⇨ `luaopen_c`
- Not intended for regular use, but helpful for some situations
  - simultaneous use of two different versions of a library

`module`

- Whole setup for a module: `module(...)`
  - create new table
  - assign it to given global name
  - assign it to `package.loaded` table
  - set it as module's environment
  - inherit for global environment
- Rest of module written like regular Lua code

Final Remarks

- Several small changes
- Incremental garbage collector should reduce pauses
  - no “real-time” garanties
- New module system should improve availability of third-part modules
  - more policies than real code
- And a last novelty...

Programming Lua, 2nd edition to be published by O'Reilly