The Novelties of Lua 5.2

Roberto Ierusalimschy

September 2011
Long list of changes

- a myriad of small improvements
- light C functions
- emergency garbage collection
- ephemerond tables
- bitlib
- yieldable pcall/metamethods
- generational collector
- goto statement
- new _ENV scheme
Light C Functions

C functions without upvalues are stored as simple values, without memory allocation
Light C Functions

- only possible due to change in environments
- new internal type
  - concept of type variant
  - opens the door for other variants (e.g., non-collectable strings)
- implemented as a single pointer to function
- eliminate the need for `lua_cpcall`
- saves a few bytes of memory
  - standard libraries create almost 200 light functions
- portable way to represent other C functions in Lua
  - C standard allows conversions between different types of C functions
Emergency Garbage Collection

when memory allocation fails, collector does a complete collection cycle and then tries again
Emergency Garbage Collection

- seems obvious, but implementation is tricky
- Lua allocates memory in lots of places
- everything must be properly anchored before any allocation
- finalizers (\_\_gc metamethods) postponed during emergency collection
Ephemeron Tables

break cycles in weak tables where values refer to their keys

typical example:

```lua
local mem = setmetatable({}, {__mode = "k"})
function newKfunc (o)
    local f = mem[o]
    if not f then
        f = function () return o end
        mem[o] = f
    end
    return f
end
```
Ephemeron Tables

- despite weak keys, entries may never be removed from mem.
  - each key has a reference to it in its value
  - values are not (and cannot be) weak
- ephemeron table: value is only alive when its key is alive
- implementation has a quadratic worst case
  - but only for weird scenarios
bitlib

library with bitwise operations
bitlib

- a most-wanted feature in Lua
- far from straightforward
  - main problem: numbers in Lua are double
  - in particular, -1 is different from 0xffffffff
- some differences from older libraries
  - signed × unsigned results
  - overflows in shift/rotate
  - negative shifts
- future problem: 64-bit operations
Yieldable `pcall/metamethods`

programs in Lua 5.2 can yield inside a `pcall`, a metamethod, or a for iterator
Yieldable pcall/metamethods

- another most-wanted feature
- planned to be the main change for Lua 5.2
- basic idea from Mike Pall
  - long-jump over C functions and call them again when resuming
  - `lua_pcall` × `lua_pcallk` allows function to signalize whether it can yield at each point
- change from original implementation: resume calls a *continuation function*
  - instead of the same function that was interrupted
  - continuation passed as argument to `lua_pcallk`
- metamethods resume through extra code to complete execution of interrupted opcodes
Generational Collector

garbage collector can use the generational algorithm
Generational Collector

- basic idea: only young objects are traversed/collected
- *infant mortality* or *generational hypothesis*
  - good: less work when traversing objects
  - bad: less memory collected
- implementation uses the same apparatus of the incremental collector
  - black objects are equated to old objects
  - black-to-white barriers become old-to-new barriers
- seems to work as expected, but with no gains in performance :(
  - hard to check without real programs
goto

Lua 5.2 will include a somewhat conventional goto statement
goto

goto fits nicely with Lua philosophy of “mechanisms instead of policies”
  - very powerful mechanism
  - easy to explain
allows the implementation of several mechanisms
  - continue, redo, break with labels, continue with labels, state machines, etc.
Yes, even break is redundant
  - may be removed in the future
  - not worth the trouble now
break does not need to be last statement in a block
  - restriction in place to allow break label in the future
  - restriction does not make sense for goto
goto implementation

- quite simple for the VM
  - small change to unify OP_CLOSE and OP_JMP
- parser must keep pending gotos and visible labels
- visibility rules
- closing of upvalues
- break implemented as goto break
  - each loop followed by a virtual label ::break::
- optimization for a common case:
  - if a == b then goto L end
  
  NEQ a b          EQ a b
  JMP 1            JMP L
  JMP L

if a == b then goto L end
Isn’t goto evil?

- “The raptor fences aren’t out are they?”
- continuations are much worse
  - dynamic and unrestricted goto
  - basic idea: \( l = \text{getlabel}(), \text{goto}(l) \)
  - labels are first-class values
- yet nobody complains; it is “cool” to support continuations
- is the problem with goto that they are too restricted?
- demonized for years of abuse
New _ENV scheme

Several parts

- _ENV instead of dynamic environments
  - any global name `var` replaced by `_ENV.var`
  - main functions receive an upvalue named `_ENV`
  - upvalue initialized with global table by default
- no more `fenv` for functions
- no more `fenv` for threads
- simplification in the support for modules
  - no more module function
modules in general, and module in particular, were the main motivations for the introduction of dynamic environments and setfenv in Lua 5.0.

module was never very popular

setfenv, on the other hand, became popular for toying with other functions

setfenv runs against function encapsulation
the new scheme, with \_ENV, allows the main benefit of setfenv with a little more than syntactic sugar
  ▶ “main benefit” being the power to encapsulate all global names of a module inside a table
being a syntactic sugar, it is \textit{much} simpler than old environments
  ▶ both implementation and proper understanding
it also allows a reasonable emulation of setfenv
  ▶ needs the \texttt{debug} library, which seems fit
as a bonus, it allows some nice tricks on its own
  ▶ \_ENV as a function argument
  ▶ setfenv bound to specific functions
Environments for C functions and threads

- environments for threads frequently misunderstood
  - only visible from C
  - when loading a new function
  - through pseudo-index for “globals”
- environments for threads seldom used
  - some few uses tricky to replace
- environments for C functions easily replaced by upvalues
- opened the door for light C functions
- less fat in the language
  - implementation and documentation
no more *module* function

in general, less implicit things

modules must explicitly change their environment and return their tables

modules do not create globals by default

- small problems with `-l` option for Lua stand-alone
- common use: `local mod = require 'mod'`
What we did not do

- removal of coercions
- macros
Automatic Coercion

- Very convenient to concatenate numbers with strings
  - `print("the value is " .. x)`
- Apparently convenient for things like `print(fact(io.read()))`
  - `function fact (n)
      if n == 0 then return 1
      else return n * fact(n - 1) end
  end`

- Mostly useless for many other cases
  - `is it?`
- Somewhat complex
Macros

- several nice solutions in the small: token filters, m4-style, etc.
- main problem (seldom discussed): programming in the large
Macros in the large

- modularization
  - what is the scope of a macro?
  - how to preload macros for a load?
- libraries providing macros
  - same library can provide both macros and functions?
  - how to “require” a library? (a predefined macro \texttt{require}?)
- how to precompile code?
  - should all macro libraries be present?
  - do macros vanish in precompiled code?
- error messages
Conclusions

- a few long-wanted features
  - yieldable pcall/metamethods
  - bitlib
  - emergency collector
- many small improvements
- good clean up of the module system
  - overdone in Lua 5.1
- there are still things to be done