Pontifícia Universidade Católica
do Rio de Janeiro

Some thorny points in the design of Lua: a personal perspective

Roberto Ierusalimschy

September 2011

the Devil is in the Details

- 0-based $\times$ 1-based arrays
- List length
- Bit library
- Goto
- Varargs
- Automatic coercion
- Numbers
- Macros

Levels of Incompatibility

Not all incompatibilities are equal!

- How frequently the incompatibility happens.
- How easily we detect the incompatibility.
- How easily we correct the incompatibility.

Detect Incompatibilities

- compilation error
- `grep`
- run-time error
- logical error

Correct Incompatibilities

- extra definition/library
  - change in a function
- fixed "macro replacement"
  - change in an operator
- local change
  - change in a control structure
- global change
  - change in a data structure

0-based × 1-based arrays

A continuous source of hatred towards Lua

All languages are 0-based; why does Lua have to be different?

Back in time . . .

- Not all languages are 0-based: Icon, Fortran, AWK[1], and Smalltalk are 1-based; Snobol, Pascal, Modula, Modula-3[2], and Ada have configurable bases.
- Currently, many languages are 0-based due to influence from C.
  - Ironically, none of them share the reason that made C 0-based (where `a[e]` means `*(a+e)`).
- However, several other languages are 0-based without that influence. Examples include Scheme, Oberon, and Haskell.

[1] When AWK creates an array for you, that array's indices are consecutive integers starting at 1.

[2] Most examples are 1-based, but open arrays start at 0.

1-based arrays

- *Much* more intuitive: first is 1st (not 0th).
  - ISO-C: "E1[E2] designates the E2-th element of E1 (counting from zero)."
- Much easier for non programmers.
- Easy for (good) programmers :)
- Historical reason: Fortran used 1-based arrays, and most first users of Lua had a Fortran background.

0-based arrays

- More interesting mathematical properties.
- Example: hash[3]: `(i%N)`
- Example: circular lists:
  - 0-based: `(i + 1)%N`, `(i - 1)%N`
  - 1-based: `i%N + 1`, `(i - 2)%N + 1`

[3] assuming a proper % operator

Antecedents

- Most languages use a `mod` operator with not-so-good mathematical properties.
  - C strikes again?
  - it does not seem to bother many people
- Lua 1.1 used degrees for trigonometric functions.
  - More intuitive for the "layman".
  - Bad mathematical properties.
  - Changed (corrected?) to radians in Lua 5.0 (!)

Change from degrees to radians

- Not too frequent
- Easy to detect
  - `grep`
- Easy to correct
  - add conversion code

Change from 1-based to 0-based

- All too frequent
- Hard to detect
  - logical errors
- Hard to correct
  - see *mod* example

Length of Lists

The crux of `#t`: Lua already has had several different mechanisms to control the length of a list.

Probably the mechanism that changed most during Lua evolution.

- intrinsic length
- extrinsic length

Intrinsic Length

- Depends only on the table itself.
- Several more-or-less useful definitions.
  - total number of elements
  - larger numerical key
  - minimal n such that ...
- Often, what should be the length is far from obvious:

```
t = {[1000] = 1}
```

- Fact: no intrinsic definition can handle lists with `nil`s at the end.

```
t = {4, 5, 10, nil, nil}
```

Extrinsic Length

- Does not depend only on the table itself.
- May depend on the "history": previous operations applied to the table
- There may be an operation `setn`.
- There *should* be an operation `setn`.
  - so that we can clone a table

`setn`

- Verbose and somewhat expensive.
  - how to add an element in a list?
- What to do with lists without a previous `setn`?
- What about constructors?
- From previous experience, an explicit use of `t.n` seems the best approach.

bitlib

- a most-wanted feature in Lua
- far from straightforward
- main problem: numbers in Lua are double
- in particular, -1 is different from 0xffffffff
- most bitwise operations not defined for non-natural numbers

bitlib

- signed × unsigned results
  - `bit.not(0) == 0xffffffff` versus `bit.not(0) == -1`
  - in Lua 5.2, all results are unsigned
- overflows in shift/rotate
  - `bit.lshift(x, 33)`
  - in Lua 5.2, all bits shifted out
- negative shifts
  - `bit.lshift(x, -33)`
  - in Lua 5.2, shift in the opposite direction
- future problem: 64-bit operations

goto

- goto fits nicely with Lua philosophy of "mechanisms instead of policies"
  - very powerful mechanism
  - easy to explain
- allows the implementation of several mechanisms
  - break, continue, redo, break with labels, continue with labels, state machines, etc.
  - Yes, even `break` is redundant

Isn't `goto` evil?

- "The raptor fences aren't out are they?"
- continuations are much worse
  - basic idea: `l = getlabel()`, `goto(l)`
  - dynamic and unrestricted `goto`
  - labels are first-class values
- yet nobody complains; it is "cool" to support continuations
- is the problem with `goto` that they are too restricted?
- Fact: more often than we want to admit, we resort to tricks to avoid the use of a `goto`

Varargs

- old-style vararg (pre-5.1): extra arguments collected in a table
  - with an `n` field!
- new-style vararg: expression '`...`' results in all extra arguments
- More efficient way to collect varargs
  - mainly to pass them to another function

Unintended consequences

- small overhead even for non-vararg functions
- demonizing table creation
  - suddenly, `{...}` becomes unacceptable
- people want to use `...` for everything
- not a good contribution to `#t`

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
    ```

Automatic Coercion

May be removed in next version.

- How frequent the incompatibility happens: should not be too frequent, but who knows?
- How easily we detect the incompatibility: medium difficulty. No syntactic method, but usually the change should result in a run-time error.
- How easily we correct the incompatibility: very easy (add explicit coercion).

Numbers

- Lua started with floats as numbers
- Changed to double in version 3.1 (1998)
  - need for 32 bits
  - bold decision at that time
- We will need 64-bit numbers; we must break the 53-bit barrier.
- Three options (at least):
  - a larger number type (e.g. `long double`)
  - more than one underlying representation
  - more than one number type

long double

- elegant solution for 64-bit machines
- too expensive for other architectures
  - not that bad with 80-bit extended precision plus the NaN trick
  - 80-bit floats give exactly 64 bits of mantissa
- Not as portable as regular Lua code

Multiple underlying representations

- example: LNUM
- Main problem: no clear arithmetic model
- Operation may give wrong result even when correct result is representable
  - `0.5 * (2^60 - 2)`

Multiple number types

- too complex
- different equal values:
  - `4294967295 == 4294967295.0`
  - `4294967295 + 1 ~= 4294967295.0 + 1`
- subtle compatibility problems

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 `require`?)
- how to precompile code?
  - should all macro libraries be present?
  - do macros vanish in precompiled code?
- error messages

the Devil is in the Details

or. . .

the Devil is in the Details

or... the color of the bike shed is not irrelevant

(by Chun Yeug Cheng and Ka Fai Lee, student competition, *Reinventing the Bike Shed*)