Small is Beautiful: the design of Lua

Roberto Ierusalimschy
PUC-Rio
Language design

- many tradeoffs
  - similar to any other design process
- designers seldom talk about them
  - what a language is not good for
Typical tradeoffs

- security x flexibility
  - static verification
- readability x conciseness
- performance x abstraction
  - specially in an interpreted language
A special tradeoff

- simplicity \times almost everything else
- several other conflicts can be solved by adding complexity
  - smarter algorithms
  - multiple mechanisms ("There's more than one way to do it")
Lua

- a scripting language
- simplicity as one of its main goals
  - small size too
- "real" language
  - many users and uses
- tricky balance between "as simple as possible" x "but not simpler"
Lua uses

• niche in games
  • "Is Lua the ultimate game scripting language?" (GDC 2010)

• embedded devices
  • cameras (Canon), keyboards (Logitech), printers (Olivetty & Océ)

• scripting applications
  • Wireshark, Snort, Nmap
Lua main goals

- simplicity/small size
- portability
- "embedability"
  - scripting!
Small size

- source lines of code (proxy for complexity)
Portability

• runs on most machines we ever heard of
  • Symbian, DS, PSP, PS3 (PPE & SPE), Android, iPhone, etc.
• written in ANSI C ∩ ANSI C++
  • avoids #ifdefs
  • avoids dark corners of the standard
Embedability

- provided as a library
- simple API
  - simple types
  - low-level operations
  - stack model
- embedded in C/C++, Java, Fortran, C#, Perl, Ruby, Python, Ada, etc.
An overview of Lua

• Conventional syntax
  • somewhat verbose

```lua
function fact (n)
    if n == 0 then
        return 1
    else
        return n * fact(n - 1)
    end
end
```

```lua
function fact (n)
    local f = 1
    for i=2,n do
        f = f * i
    end
    return f
end
```
An overview of Lua

- semantically quite similar to Scheme
- dynamically typed
- functions are first-class values with static scoping
BTW...

```lua
function fact (n)
    local f = 1
    for i=2,n do f = f * i; end
    return f
end
```

```lua
fact = function (n)
    local f = 1
    for i=2,n do f = f * i; end
    return f
end
```

syntactic sugar
An overview of Lua

- proper tail recursive
- Lua does not have full continuations, but have one-shot continuations
  - in the form of coroutines
Design

- tables
- coroutines
- the Lua-C API
Tables

- associative arrays
  - any value as key
- only data-structure mechanism in Lua
Why tables

- VDM: maps, sequences, and (finite) sets
  - collections
- any one can represent the others
- only maps represent the others with simple \textit{and} efficient code
Data structures

- tables implement most data structures in a simple and efficient way
- records: syntactical sugar `t.x` for `t["x"]`:

```python

t = {}
t.x = 10
t.y = 20
print(t.x, t.y)
print(t["x"], t["y"])
```
Data Structures

• arrays: integers as indices

\[ a = \{\} \]
\[ \text{for } i=1,n \text{ do } a[i] = 0 \text{ end} \]

• sets: elements as indices

\[ t = \{\} \]
\[ t[x] = \text{true} \quad \text{-- } t = t \cup \{x\} \]
\[ \text{if } t[x] \text{ then } \quad \text{-- } x \in t? \]
\[ \ldots \]
Other constructions

- tables also implement modules
  - print(math.sin(3))
- tables also implement objects
  - with the help of a delegation mechanism and some syntactic sugar
Objects

- first-class functions + tables \(\approx\) objects
- syntactical sugar for methods
  - handles self

\[
\begin{align*}
\text{function } a:foo (x) \\
& \quad \ldots \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
a:foo(x) \rightarrow a.foo(a, x)
\end{align*}
\]

\[
\begin{align*}
a.foo = \text{function } (\text{self}, x) \\
& \quad \ldots \\
\text{end}
\end{align*}
\]
Delegation

- field-access delegation (instead of method-call delegation)
- when a delegates to b, any field absent in a is got from b
  - a[k] becomes (a[k] or b[k])
- allows prototype-based and class-based objects
- allows single inheritance
Delegation at work

```javascript
a:
k = 0
delegate:

"class":
foo = function ...

a:foo(x)  →  a.foo(a,x)
```
Tables: problems

- The implementation of a concept with tables is not as good as a primitive implementation
  - Access control in objects
  - Length in sequences
- Different implementations confound programmers
  - DIY object systems
Coroutines

- old and well-established concept, but with several variations
- variations not equivalent
  - several languages implement restricted forms of coroutines that are not equivalent to one-shot continuations
Coroutines in Lua

c = coroutine.create(function ()
  print(1)
  coroutine.yield()
  print(2)
end)

coroutine.resume(c)  -->  1
coroutine.resume(c)  -->  2
Coroutines in Lua

- first-class values
  - in particular, we may invoke a coroutine from any point in a program
- stackful
  - a coroutine can transfer control from inside any number of function calls
- asymmetrical
  - different commands to resume and to yield
Coroutines in Lua

- simple and efficient implementation
  - the easy part of multithreading
- first class + stackful = complete coroutines
  - equivalent to one-shot continuations
  - we can implement call/1cc
- coroutines present one-shot continuations in a format that is more familiar to most programmers
Coroutines x continuations

- most uses of continuations can be coded with coroutines
  - "who has the main loop" problem
    - producer-consumer
    - extending x embedding
  - iterators x generators
    - the same-fringe problem
  - collaborative multithreading
Coroutines x continuations

- multi-shot continuations are more expressive than coroutines
- some techniques need code reorganization to be solved with coroutines or one-shot continuations
  - oracle functions
The Lua-C API

- Lua is a library
  - formally, an ADT (a quite complex one)
  - 79 functions
- the entire language actually describes the argument to one function of that library: `load`
  - `load` gets a stream with source code and returns a function that is semantically equivalent to that code
The Lua-C API

- most APIs use some kind of "Value" type in C
  - \texttt{PyObject} (Python), \texttt{jobject} (JNI)
- problem: garbage collection
  - Python: explicit manipulation of reference counts
  - JNI: local and global references
- too easy to create dangling references and memory leaks
The Lua-C API

- Lua API has no "LuaObject" type
- a Lua object lives only inside Lua
- two structures keep objects used by C:
  - the stack
  - the registry
The Stack

- keep all Lua objects in use by a C function
- injection functions
  - convert a C value into a Lua value
  - push the result into the stack
- projection functions
  - convert a Lua value into a C value
  - get the Lua value from anywhere in the stack
The Stack

• example: calling a Lua function from C

/* calling f("hello", 4.5) */
lua_getglobal(L, "f");
lua_pushstring(L, "hello");
lua_pushnumber(L, 4.5);
lua_call(L, 2, 1);
if (lua_isnumber(L, -1))
    printf("%f\n", lua_getnumber(L, -1));
The Stack

- example: calling a Lua function from C
  - push function

```c
/* calling f("hello", 4.5) */
lua_getglobal(L, "f");
lua_pushstring(L, "hello");
lua_pushnumber(L, 4.5);
lua_call(L, 2, 1);
if (lua_isnumber(L, -1))
    printf("%f
", lua_getnumber(L, -1));
```
The Stack

- example: calling a Lua function from C
  - push function, push arguments,
The Stack

- example: calling a Lua function from C
  - push function, push arguments, do the call

```c
/* calling f("hello", 4.5) */
lua_getglobal(L, "f");
lua_pushstring(L, "hello");
lua_pushnumber(L, 4.5);
lua_call(L, 2, 1);
if (lua_isnumber(L, -1))
    printf("%f\n", lua_getnumber(L, -1));
```
The Stack

- example: calling a Lua function from C
  - push function, push arguments, do the call, get result from the stack

```c
/* calling f("hello", 4.5) */
lua_getglobal(L, "f");
lua_pushstring(L, "hello");
lua_pushnumber(L, 4.5);
lua_call(L, 2, 1);
if (lua_isnumber(L, -1))
    printf("%f\n", lua_getnumber(L, -1));
```
The Stack

- example: calling a C function from Lua

```c
static int l_sqrt (lua_State *L) {
    double n = luaL_checknumber(L, 1);
    lua_pushnumber(L, sqrt(n));
    return 1;  /* number of results */
}
```
The Stack

- example: calling a C function from Lua
  - get arguments from the stack

```c
static int l_sqrt (lua_State *L) {
    double n = luaL_checknumber(L, 1);
    lua_pushnumber(L, sqrt(n));
    return 1;  /* number of results */
}
```
The Stack

• example: calling a C function from Lua
  • get arguments from the stack, do computation

```
static int l_sqrt (lua_State *L) {
    double n = luaL_checknumber(L, 1);
    lua_pushnumber(L, sqrt(n));
    return 1;  /* number of results */
}
```
The Stack

• example: calling a C function from Lua
  • get arguments from the stack, do computation, push arguments into the stack

```c
static int l_sqrt (lua_State *L) {
    double n = luaL_checknumber(L, 1);
    lua_pushnumber(L, sqrt(n));
    return 1; /* number of results */
}
```
The Registry

- sometimes, a reference to a Lua object must outlast a C function
  - `NewGlobalRef` in the JNI
- the *registry* is a regular Lua table always accessible by the API
  - no new concepts
  - to create a new "global reference", store the Lua object at a unique key in the registry and keeps the key
The Lua-C API: problems

- too low level
  - some operations need too many calls
- stack-oriented programming sometimes is confusing
  - what is where
- no direct mapping of complex types
  - may be slow for large values
Conclusions

• any language design involves conflicting goals

• designers must solve conflicts
  • consciously or not

• to get simplicity we must give something
  • performance, easy of use, particular features or libraries,
Conclusions

- simplicity is not an absolute goal
- it must be pursued incessantly as the language evolve
- it is much easier to add a feature than to remove one
  - start simple, grow as needed
- it is very hard to anticipate all implications of a new feature
  - clash with future features
Conclusions

- "Mechanisms instead of policies"
  - e.g., delegation
  - effective way to avoid tough decisions
  - this itself is a decision...