Small is Beautiful: the Design of Lua

Roberto Ierusalimschy

Pontifícia Universidade Católica do Rio de Janeiro

Language design

- many tradeoffs
  - similar to any other design process
- designers seldom talk about them
  - what a language is not good for
  - the myth of the general purpose language
- we need explicit goals to solve tradeoffs

Typical tradeoffs

- security x flexibility
  - static verification
- compile time x run time
- readability x conciseness
- performance x abstraction
  - specially in an interpreted language
- readability x readability
  - to whom?

Real-world tradeoffs

- conciseness x good error messages
- flexibility x good error messages
- flexibility x strong community
- evolution x “general knowledge”
- good libraries x portability

A special tradeoff

- simplicity x *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
- tricky balance between “as simple as possible” x “but not simpler”
- many users and uses

embedded devices

- TVs (Samsung), routers (Cisco), keyboards (Logitech), printers (Olivetti), set-top boxes (Verizon), M2M devices (Sierra Wireless), calculators (TI-Nspire), scripting applications

- Wireshark, Snort, Nmap, VLC Media Player, LuaTeX, …

`http://en.wikipedia.org/wiki/Category:Lua-scriptable_software`

Slashdot: News for nerds, Feb 1, 2012:

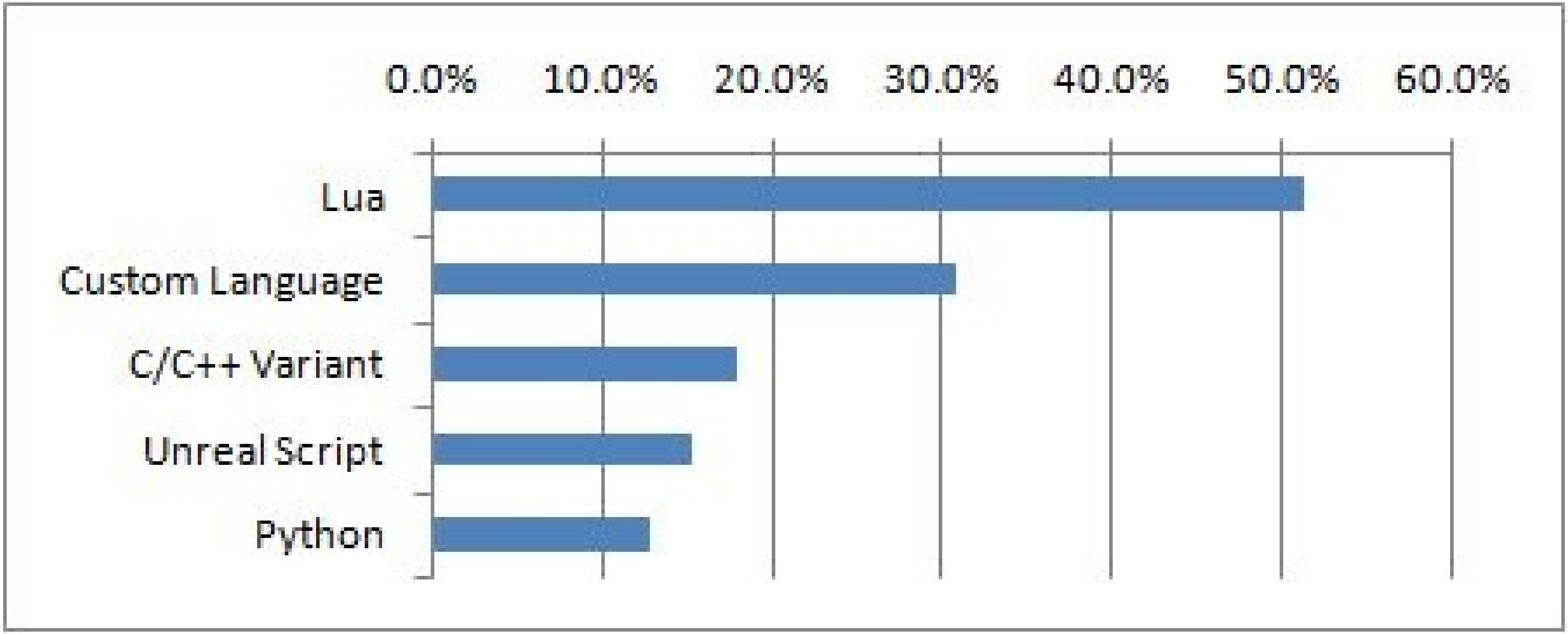
"Wikipedia Chooses Lua As Its New template language"

Adobe Lightroom

One million lines of Lua code

Lua in games

The Engine Survey (03/02/09,Gamasutra):
What script languages are most people using?




Company of Heroes
Universe at War
Dungeons
Tap Tap Revenge
Blitzkrieg
Introducing The World's First Competitive Sim!
Monopoly Tycoon
Driver San Francisco
S.T.A.L.K.E.R.
Grim Fandango
Amped
Hulk
Destroy All Humans!
Baldur's Gate
World of Warcraft
Far Cry
Impossible Creatures
NFL Fever
Soul Ride
Diner Dash 2
Angry Birds
Play
SpellForce
Napoleon Total War
Supreme Commander
Fable II
Buzz! Quiz TV
Brütal Legend
SimCity 4
Topspin 2
Sonic the Hedgehog
Star Wars Empire at War
The Guild 2
MDK2
Crysis
Supreme Commander Forged Alliance
Homeworld2
Psychonauts
The Incredibles
When Danger Calls
The Witcher
Crackdown
Links
Empire Total War
Escape from Monkey Island
Civilization V
plus+
Heroes V
Dawn of War Dark Crusade
Bubble
Ball

Lua main goals

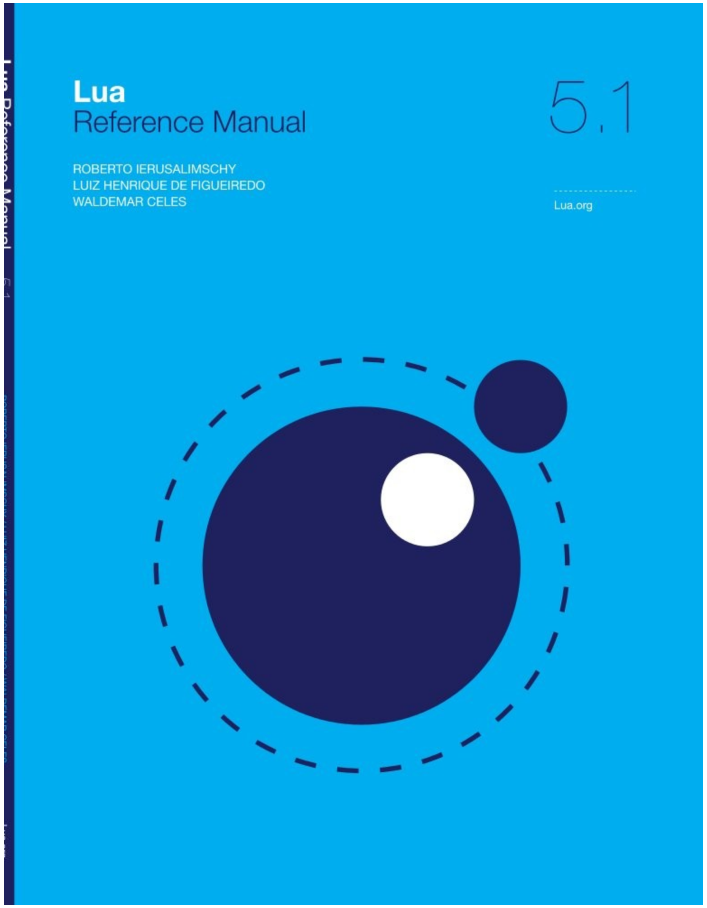
- simplicity
- small size
- portability
- embedability
  - scripting!

Simplicity

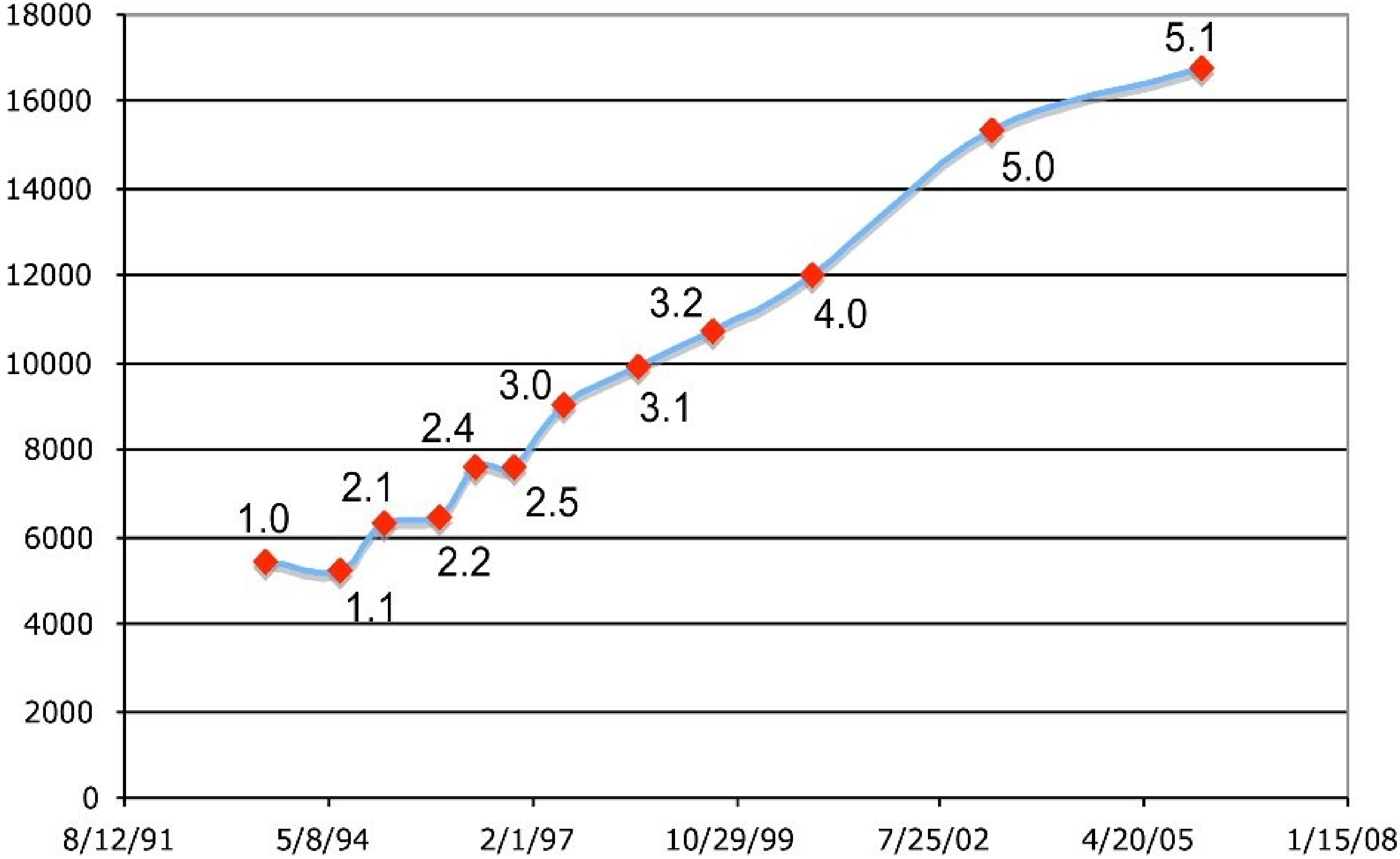
Reference manual with 100 pages (proxy for complexity)

documents language, libraries, and C API

(spine)



Small size



18000
16000
14000
12000
10000
8000
6000
4000
2000
0
8/12/91
5/8/94
2/1/97
10/29/99
7/25/02
4/20/05
1/15/08
1.0
1.1
2.1
2.2
2.4
2.5
3.0
3.1
3.2
4.0
5.0
5.1

Portability

- runs on most platforms we ever heard of
  - Symbian, Nintendo DS, PSP, PS3 (PPE & SPE), Android, iOS, IBM z/OS, etc.
- runs inside OS kernels
  - FreeBSD, Linux
- written in ANSI C $\cap$ ANSI C++
  - avoids `#ifdefs`
  - avoids dark corners of the C standard

Embedability

- Emphasis on scripting
  - to be used together with a *system language*
  - *tight integration between languages*
  - *not only external libraries*
- Provided as a library
- Not only an implementation issue
- Embedded in C/C++, Java, Fortran, C#, Perl, Ruby, Python, etc.

An overview of Lua

- Conventional syntax
  - somewhat verbose

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

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

An overview of Lua

- semantically somewhat similar to Scheme
- similar to JavaScript, too
  - Lua predates JS by two years
- dynamically typed
- all objects have unlimited extent
  - incremental garbage collector
- functions are first-class values with static scoping
- proper tail recursive

BTW...

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

syntactic sugar

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

An overview of Lua

- numbers are doubles
- 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 *and* efficient code

Data structures

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

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

Data Structures

- arrays: integers as indices

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

- sets: elements as indices

```
t = {}
t[x] = true       -- t = t ∪ {x}
if t[x] then      -- x ∈ t?
  ...
```

Modules

- Tables populated with functions

```
local math = require "math"
print(math.sqrt(10))
```

- Several facilities come for free
  - submodules
  - local names

```
local m = require "math"
print(m.sqrt(20))
local f = m.sqrt
print(f(10))
```

Objects

- first-class functions + tables ≈ objects
- syntactical sugar for methods
  - handles self

```
a:foo(x)
```

```
a.foo(a,x)
```

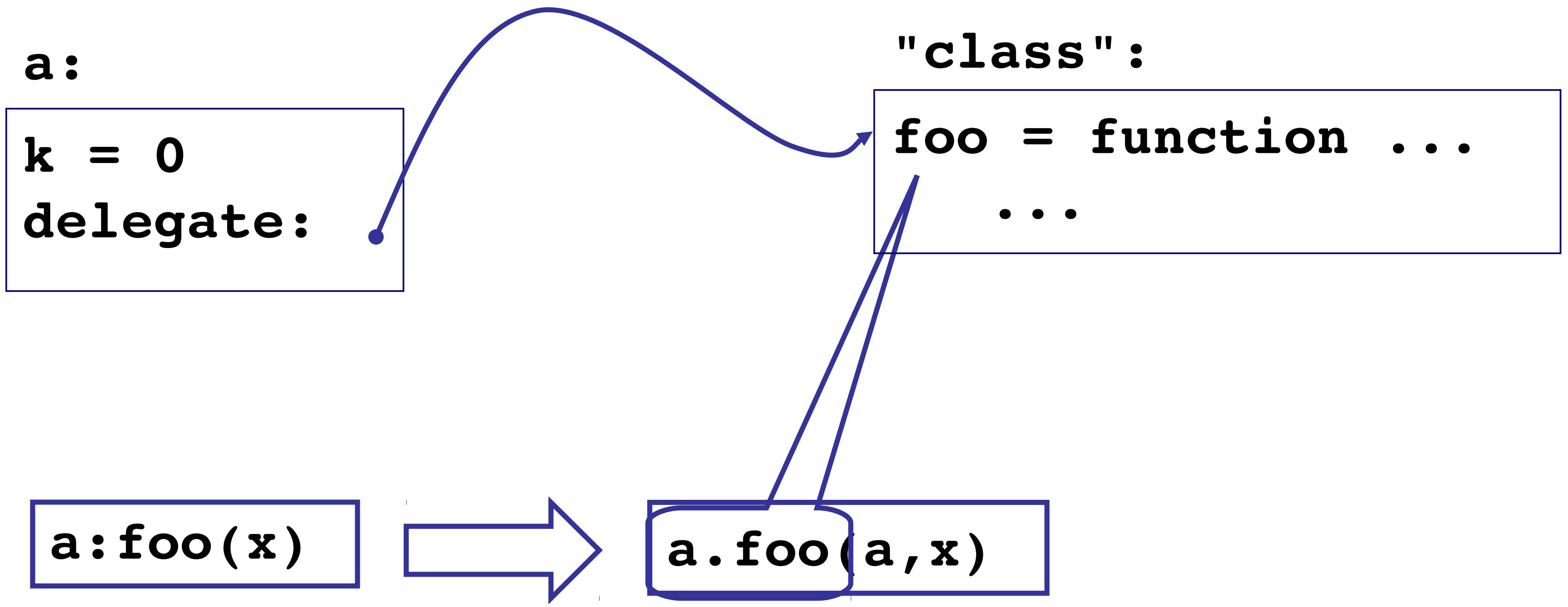
```
function a:foo (x)
  ...
end
```

```
a.foo = function (self,x)
  ...
end
```

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



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

Asymmetric coroutines

- asymmetric and symmetric coroutines are equivalent
- not when there are different kinds of contexts
  - integration with C
- how to do a **`transfer`** with C activation records in the stack?
- **`resume`** fits naturally in the C API

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

Basic (Naive) Lua Interpreter

```
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  luaL_loadfile(L, argv[1]);
  lua_call(L, 0, 0);
  return 0;
}
```

The Lua-C API

- most APIs use some kind of “Value” type in C
  - **`PyObject`** (Python), **`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 registry
  - the stack

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 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
  - push function, push arguments, do the call, get result from the stack

```
/* 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
  - get arguments from the stack, do computation, push arguments into the stack

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

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...

Trade-offs