
# Small is Beautiful: the design of Lua

Roberto Ierusalimschy

PUC-Rio

# 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
```

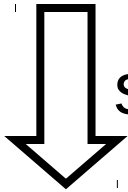
```
function fact (n)
  local f = 1
  for i=2,n do
    f = f * i
  end
  return f
end
```

# An overview of Lua

- semantically somewhat similar to Scheme
  - follows Section 1.1, Semantics, of Revised (5) report on Scheme, except for continuations and numbers
- dynamically typed
- all objects have unlimited extent
- 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

# Tables

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

# Why tables

- VDM: maps, sequences, and (finite) sets
- 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?
   ...
```

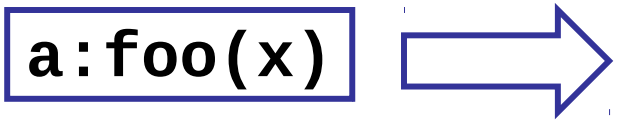
# Other constructions

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

# Objects

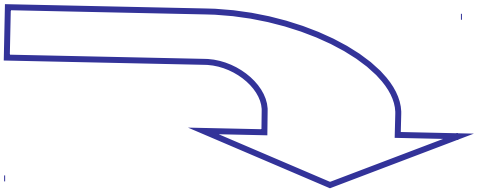
- first-class functions + tables ≈ objects
- syntactical sugar for methods
  - handles self

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



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

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

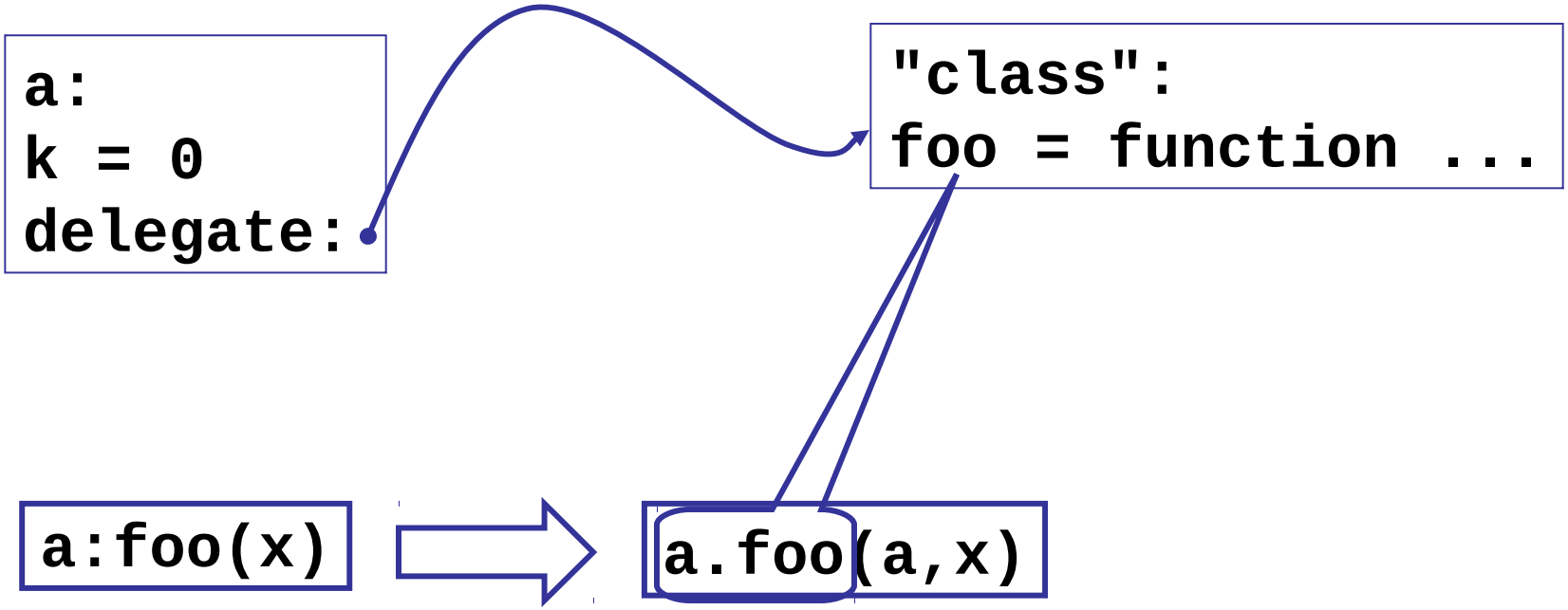


```
a.foo = function (self,x)
  ...
end
```

# 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
- asymmetric
  - 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

# Conclusions

- to get simplicity we must give something
  - performance, easy of use, particular features, libraries
- 
- “Mechanisms instead of policies”
  - e.g., OO model
  - effective way to avoid tough decisions
  - this itself is a decision...


**www.lua.org**