Programming in Lua – Metatables

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

Metatables

- A *metatable* modifies the behavior of another table; by setting a metatable with appropriate fields, you can:
  - Use arithmetic, concatenation, and relational operators
  - Override the behavior of ==, ~=, and # operators
  - Override the behavior of the tostring, pairs, and ipairs built-in functions
  - Provide values for missing fields and intercept the creation of new fields
  - Call a table as a function

Scope of metatables

- Each table can have its own metatable, which will modify just the behavior of that single table
- But several tables can share a single metatable, so they will all have similar behavior → the usual application: create new data types
- The built-in function `setmetatable` changes the metatable of a table, and returns the table
- The built-in function `getmetatable` returns the metatable of a table (or nil if it does not have one)
- It is not good programming style to modify a metatable after assigning it to a table, as this may impact performance

Metamethods

- You specify the operations that a metatable will modify by setting *metamethods*

- A metamethod is a function associated with a specially named field

two underscores

- There are 19 metamethods: `__add, __sub, __mul, __div, __mod, __pow, __unm, __concat, __len, __eq, __lt, __le, __index, __newindex, __call, __tostring, __ipairs, __pairs, __gc`

- Almost all metamethods must be functions, except for `__index` and `__newindex`, which can also be tables; using a table for `__index` is the basis of single-inheritance OO programming in Lua

Complex numbers

- As a motivating example, we will use metamethods to augment the complex numbers of unit 9 – Modules with several operations:
  - Addition to reals and other complex numbers with `+` (the same techniques will work for the other arithmetic operations)
  - Structural comparison for equality (two complex numbers are equal their real and imaginary parts are equal)
  - Modulus with `#`
  - Pretty-printing with `tostring`

Sharing a metatable

- We first create a table private to the module and set it as the metatable for each complex number we create with new:

```
local mt = {}

local function new(r, i)
  return setmetatable({ real = r or 0, im = i or 0 }, mt)
end
```

- This metatable gives us a nice test to see if an arbitrary value is a complex number or not:

```
local function is_complex(v)
  return getmetatable(v) == mt
end
```

Overloading + with __add

- The add function already adds two complex numbers; if we assign it to the __add field of the metatable, + will begin working with a pair of complex numbers:

```
local function add(c1, c2)
  return new(c1.real + c2.real, c1.im + c2.im)
end

mt.__add = add
```

```
> c1 = complex.new(2, 3)
> c2 = complex.new(1, 5)
> print(complex.tostring(c1 + c2))
3+8i
```

- Let us see what happens when we add a real to a complex:

```
> c3 = c1 + 5
.\complex.lua:20: attempt to index local 'c2' (a number value)
stack traceback:
        .\complex.lua:20: in function '__add'
        stdin:1: in main chunk
        [C]: in ?
```

Arithmetic resolution

- What is happening? Lua is calling the `__add` metamethod of the complex number! If the left operand has an `__add` metamethod Lua *will* call it. We can take advantage of that:

```
local function add(c1, c2)
  if not is_complex(c2) then
    return new(c1.real + c2, c1.im)
  end
  return new(c1.real + c2.real, c1.im + c2.im)
end
```

- Now adding a real to a complex works:

```
> c1 = complex.new(2, 3)
> c3 = c1 + 5
> print(complex.tostring(c3))
7+3i
```

Arithmetic resolution (2)

- What about adding a complex to a real?

```
> c3 = 5 + c1
.\complex.lua:20: attempt to index local 'c1' (a number value)
stack traceback:
        .\complex.lua:20: in function '__add'
        stdin:1: in main chunk
        [C]: in ?
```

- If the left operand does not have a metamethod and the second has, Lua will call the metamethod of the second operand! This gives us the final form of add:

```
local function add(c1, c2)
  if not is_complex(c1) then
    return new(c2.real + c1, c2.im)
  end
  if not is_complex(c2) then
    return new(c1.real + c2, c1.im)
  end
  return new(c1.real + c2.real, c1.im + c2.im)
end
```

```
> c3 = 5 + c1
> print(complex.tostring(c3))
7+3i
```

Equality

- The metamethod `__eq` controls both == and ~=

- It follows slightly different rules from arithmetic, as Lua will only call the metamethod if both operands have the same metatable. This gives us a simple implementation of equality for complex numbers:

```lua
local function eq(c1, c2)
  return (c1.real == c2.real) and (c1.im == c2.im)
end

mt.__eq = eq
```

```
> c1 = complex.new(1, 2)
> c2 = complex.new(2, 3)
> c3 = complex.new(3, 5)
> print(c1 + c2 == c3)
true
> print(c1 ~= c2)
true
```

- The disadvantage is that comparisons of complex numbers and reals will always be false, even if the imaginary part is zero

Overloading # and tostring

- Both the __len and __tostring metamethods work in a similar way: they receive the table and should return their result; this makes adding them to our complex numbers straightforward:

```
local function modulus(c)
  return math.sqrt(c.real * c.real + c.im * c.im)
end

mt.__len = modulus

local function tos(c)
  return tostring(c.real) .. "+" .. tostring(c.im) .. "i"
end

mt.__tostring = tos
```

```
> c1 = complex.new(3, 4)
> print(#c1)
5
> print(tostring(c1))
3+4i
> print(c1)
3+4i
```

- print uses tostring

Relational operations

- The metamethod for `<` (`__lt`) works just like the arithmetic metamethods; for `>`, lua uses `__lt` with the operands reversed

- The metamethod for `<=` (`__le`) also works like an arithmetic metamethod, but `<=` will use `__lt` if `__le` is not available, reversing the operands and negating

- Why two metamethods, then? For *partial orders*:

```lua
local function le(c1, c2)
  if not is_complex(c1) then
    return (c1 <= c2.real) and (c2.im >= 0)
  end
  if not is_complex(c2) then
    return (c1.real <= c2) and (c2.im <= 0)
  end
  return (c1.real <= c2.real) and (c1.im <= c2.im)
end

mt.__le = le
```

```lua
local function lt(c1, c2)
  return c1 <= c2 and not (c2 <= c1)
end

mt.__lt = lt
```

`__index` and `__newindex`

- If the metatable has an `__index` metamethod Lua will call it, passing the table and the key, whenever the key cannot be found; what the metamethod returns is the result of the indexing operation

- If the metatable has a `__newindex` metamethod Lua will call it, passing the table, the key and the value, whenever Lua is assigning to a key that is not present

- A common application of both metamethods is to use them in concert with an an empty table to act as a *proxy* for another table; the proxy is kept empty so all indexing operations are intercepted

- Both `__index` and `__newindex` can be tables instead of functions; in this case Lua will redo the indexing operation on the this table

A counting proxy

```
local mt = {}

function mt.__index(t, k)
  t.__READS = t.__READS + 1
  return t.__TABLE[k]
end

function mt.__newindex(t, k, v)
  t.__WRITES = t.__WRITES + 1
  t.__TABLE[k] = v
end

local function track(t)
  local proxy = { __TABLE = t, __READS = 0, __WRITES = 0}
  return setmetatable(proxy, mt)
end

return { track = track }
```

```
> proxy = require "proxy"
> t = proxy.track({})
> t.foo = 5
> print(t.foo)
5
> t.foo = 2
> print(t.__READS, t.__WRITES)
1       2
```

Quiz

- We can try to work around the limitation of `__eq` so we can have `complex.new(2,0) == 2` by making complex.new return a real if the imaginary part is 0. Which operations will continue to work with this change, and which will not work anymore?