

An overview of Lua

Julia Lawall
DIKU

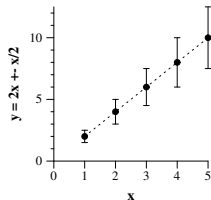
February 9, 2006

Pre-history (1992)

- ▶ DEL (data-entry language):
 - Language for structuring graphical front-ends for simulation data
 - Luiz Henrique de Figueiredo, Luiz Cristovüao Gomes Coelho, et al. at the Pontifical Catholic University of Rio de Janeiro, Brazil.
- ▶ Sol (Simple Object Language):
 - Language for translating from human readable report descriptions to those understood by a legacy tool
 - Roberto Ierusalimsky, et al. at the Pontifical Catholic University of Rio de Janeiro, Brazil.
- ▶ Issues:
 - Computing values from other values.
 - Putting constraints on values.
 - Communication with C.

More concretely: the jgraph graph language

One possibility:



```
newgraph
axis min 0 max 5 size 1.5 mhash 0 label : x
yaxis min 0 max 12.5 size 1.5 label : y = 2x +/- x/2
```

```
newcurve marktype circle linetype dotted
y_epts
  1 2 1.5 2.5      2 4 3 5          3 6 4.5 7.5
  4 8 6 10        5 10 7.5 12.5
```

Another possibility:

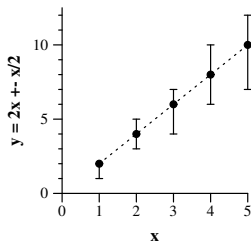
```
newgraph
axis min 0 max 5 size 1.5 mhash 0 label : x
yaxis min 0 max 12.5 size 1.5 label : y = 2x +/- x/2
```

```
newcurve marktype circle linetype dotted
y_epts shell : echo "" |\
awk '{ for (i = 1; i < 6; i++) \
      print i, 2*i, 2*i-i/2.0, 2*i+i/2.0}'
```

A problem

```
newgraph
xaxis min 0 max 5 size 1.5 mhash 0 label : x
yaxis min 0 max 12.5 size 1.5 label : y = 2x +- x/2

newcurve marktype circle linetype dotted
y_epts shell : echo "" | \
awk '{ for (i = 1; i < 6; i++) print i, 2*i, 2*i-i/2.0, 2*i+i/2.0}'
```



Lua (1993-)

Developers

- ▶ Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes.
- ▶ Computer graphics technology group, Pontifical Catholic University of Rio de Janeiro, Brazil.
- ▶ “the only language developed outside the industrialized world to have achieved global relevance.”

Goals

- ▶ Simple syntax, small implementation.
- ▶ Extensibility.
- ▶ Fast compilation.
- ▶ Easy interaction with C.

Some examples

A simple example

```
width = 420
height = width * 3/2
color = "blue"
```

A more complex example

```
function Bound(w, h)
  if w < 20 then w = 20
  elseif w > 500 then w = 500
  end
  local minH = w*3/2
  if h < minH then h = minH end
  return w, h
end
```

```
width, height = Bound(420, 500)
if monochrome then color = "black" else color = "blue" end
```

Some language features

- ▶ Tables (associative arrays).
- ▶ Fallbacks.
- ▶ Functions.
- ▶ Interaction with C.
- ▶ Interaction with ML.

Tables

Lua 1.1 (1994)

- ▶ @()
- ▶ @[2, 4, 9, 16, 25]
- ▶ @{name = "John", age = 35}
- ▶ @foo[2, 4, 9], @bar{name = "John", age = 35}

Lua 2.1 (1995)

- ▶ {}
- ▶ {2, 4, 9, 16, 25}
- ▶ {name = "John", age = 35}
- ▶ {2, 4, 9; name = "John", age = 35}

Lua 5.0 (2003)

- ▶ {2, 4, name = "John", age = 35, 9}

Using tables

```
> info = {2, 4, name = "John", age = 35, 9}
```

```
> = info[1], info[2], info[3]
```

```
2 4 9
```

```
> = info.name, info.age
```

```
John 35
```

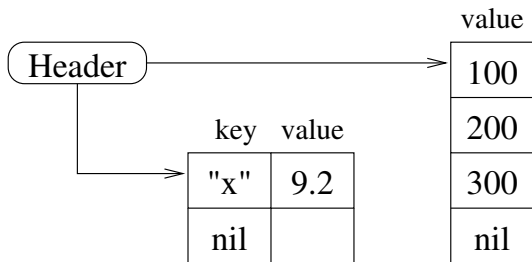
```
> john = info
```

```
> john.birthdate = 2005 - 35
```

```
> = info.birthdate
```

```
1970
```

Table implementation



Fallbacks: error handling/language extension

Lua 1.1

- ▶ One hook for handling all errors.

Lua 2.1

- ▶ Host program can set a handler for each kind of error.
- ▶ Examples: “arith”, “index”, “gettable”, “function”

Lua 3.0

- ▶ Tags and tag methods.
- ▶ Apply to all objects with a specific tag.

Lua 5.0

- ▶ Metatables and metamethods.
- ▶ Attached to an individual object.

Implementing inheritance (Lua 2.1)

```
function Index (a,i)
  if i == "parent" then                -- avoid loops
    return nil
  end
  local p = a.parent
  if type(p) == "table" then
    return p[i]                        -- may trigger Index again
  else
    return nil
  end
end
setfallback("index", Index)          -- set Index as a fallback

a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400, parent=a}
print(b.color)
```

Implementing inheritance (Lua 3.0)

```
function Index (a,i)
  if i == "parent" then
    return nil
  end
  ...
end

-- set Index as a fallback
inherit_by_parent = newtag()
settagmethod(inherit_by_parent,"index", Index)

a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400, parent=a}
settag(b,inherit_by_parent)
print(b.color)
```

Implementing inheritance (Lua 5.0)

```
function Index (a,i)
  if i == "parent" then
    return nil
  end
  ...
end
```

```
a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400, parent=a}
setmetatable(b, __index = Index)
print(b.color)
```

Alternatively:

```
a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400}
setmetatable(b, __index = a)
print(b.color)
```

Functions

Lua functions are first-class values. They can be:

- ▶ Passed as arguments
- ▶ Stored in tables
- ▶ Returned from functions
- ▶ Created at run time, since Lua 3.1

Implementing first-class functions

```
function fn ()  
  local x = 3  
  local returned = function () return x end  
  x = 27  
  return returned  
end
```

```
new_fn = fn()  
print(new_fn())
```

- ▶ Normally, `local x` is stored on the stack. That location is invalid when `new_fn` is called.
- ▶ The creation-time value of `x` cannot be used, because of `x = 27`.

Lisp solution

```
(define (fn)
  (let* ([x 3]
         [returned (lambda () x)])
    (set! x 27)
    returned))

(define new_fn (fn))
(let ([x 452]) (printf "~a~n" (new_fn)))
```

- ▶ Variables get their value from their most recent definition at runtime.
- ▶ Result is 452.
- ▶ Difficult to understand and easy to abuse.

ML solution

```
let fn() =  
  let x = ref 3 in  
  let returned = function () -> !x in  
  x := 27;  
  returned  
  
let new_fn = fn()  
let _ = Printf.printf "%d\n" (new_fn())
```

- ▶ The value of `x` is a heap location, which persists.
- ▶ This value is copied into the closure.
- ▶ The value is still valid when `new_fn` is called.
- ▶ The hard work is done by the programmer!

Scheme solution

```
(define (fn)
  (let* ([x 3]
         [returned (lambda () x)])
    (set! x 27)
    returned))
```

```
(define new_fn (fn))
(sprintf "~a~n" (new_fn))
```

- ▶ Scan once to find the variables that can be stored on the stack and the variables that must be stored in the heap.
- ▶ Construct ML-like boxing and unboxing accordingly.
- ▶ Larger, slower, two-pass compiler!
- ▶ “the cfa of Bigloo ... is more than ten times larger than the whole Lua implementation”

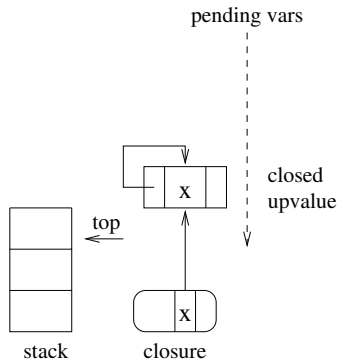
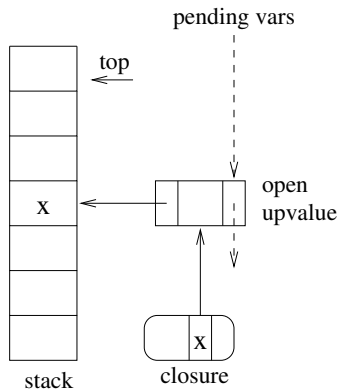
Lua 3.1 solution

```
function fn ()
  local x = 3
  local returned = function () return %x end
  x = 27
  return returned
end

new_fn = fn()
print(new_fn())
```

- ▶ Creation-time value of `x` is stored in the closure.
- ▶ `print(new_fn())` prints 3.
- ▶ Confusing, less expressive.

Lua 5.0 solution



Interaction with the host program

Issues:

- ▶ Using Lua values in host-language code.
- ▶ Using host functions in Lua.
- ▶ Using host values in Lua code.
 - Host values have the Lua type userdata.
- ▶ Using Lua functions in the host program.
 - Invoke the Lua interpreter.

C API: Using Lua values in the host-language code

Lua values remain on a stack or other internal storage.

- ▶ Implementation (inaccessible to the host program):

⟨tag, value⟩

- ▶ Project: `lua_tonumber`
 - extracts a float from a Lua value
- ▶ Test type: `lua_isnumber`
 - tests whether a Lua value represents a float
- ▶ Embed: `lua_pushnumber`
 - provides a float to Lua

C API: Using host functions in Lua

```
void my_atan(void) {
    if (!lua_isnumber(lua_getparam(1)))
        lua_error("first arg not a number");
    if (!lua_isnumber(lua_getparam(2)))
        lua_error("second arg not a number");
    lua_pushnumber(
        atan2(lua_getnumber(lua_getparam(1)),
              lua_getnumber(lua_getparam(2))));
}

lua_register("atan",my_atan);
```

It would be nice to encapsulate this glue code in a function, but not possible in C.

Another example

Lua function

```
function foo(t)
  return t.x
end
```

C version: Lua 1.0

```
typedef struct Object *lua_Object;    // ptr into Lua memory

void foo_l (void) {
  lua_Object t = lua_getparam(1);
  lua_Object r = lua_getfield(t, "x");
  lua_pushobject(r);
  return;
}
```

Problem: lua_Object value can be unexpectedly modified by Lua API functions.

The example in Lua 2.1

Values for use in C code copied to a separate storage space.

```
typedef unsigned int lua_Object;    // index into the storage space

void foo_1 (void) {
    lua_Object t = lua_getparam(1);
    lua_pushobject(t);
    lua_pushstring("x");
    lua_Object r = lua_getsubscript();
    lua_pushobject(r);
    return;
}
```

Problem: Values stay in the storage space until the C function completes. Possible overflow.

The example in Lua 4.0

The flat storage space is replaced by a stack.

```
int foo_l (lua_State *L) {  
    lua_pushstring(L, "x");  
    lua_gettable(L, 1);  
    return 1;          // the number of values to return to Lua  
}
```

ML API: Using Lua values in the host-language code

Lua values are ML values:

```
type value =  
  Nil  
  | Number of float  
  | String of string  
  | Function of srcloc * funty  
  | Userdata of userdata  
  | Table of table  
and funty = value list -> value list  
and table = (value, value) Luahash.t
```

Values can be extracted from the Lua state directly, or accessed via functions.

Embedding and projection

For each base type there are embed and project functions to convert from and to Lua values

- ▶ Embed for float:

```
function x -> Number x
```

- ▶ Project for float:

```
function  
  Number x -> x  
  | String s when is_float_literal s -> float_of_string s  
  | v -> raise (Projection (v, "float"))
```

Functions are more complicated, because of arity issues.

ML API: Using host functions in Lua

```
let my_atan2 =  
  let f = func (float **-> float **-> result float) in  
    f.embed atan2
```

Lua and game programming

Advantages of Lua:

- ▶ Portability.
- ▶ Efficiency and small size.
- ▶ Ease of embedding: Lua and C/C++ can each be used where appropriate.
- ▶ Simplicity.
- ▶ Stability.
- ▶ Convenient license.
- ▶ Coroutines, procedural representation of data.

Lua developers did not target game programming.

Summary

Goals

- ▶ Simple syntax, small implementation.
- ▶ Extensibility.
- ▶ Fast compilation.
- ▶ Easy interaction with C.

Features

- ▶ Tables, fallbacks, functions.

Language embedding

- ▶ Interacts with C, C++, and ML.

Literature

- ▶ R. Ierusalimschy, L. H. de Figueiredo, W. Celes, “Lua 5.0 Reference Manual,” Technical Report MCC-14/03, PUC-Rio, 2003.
- ▶ R. Ierusalimschy, L. H. de Figueiredo, W. Celes, “Lua-an extensible extension language,” *Software: Practice & Experience* 26 #6 (1996) 635-652.
- ▶ R. Ierusalimschy, L. H. de Figueiredo, W. Celes, “The implementation of Lua 5.0,” *Journal of Universal Computer Science* 11 #7 (2005) 1159-1176. Also *Proceedings of IX Brazilian Symposium on Programming Languages* (2005) 63-75.
- ▶ R. Ierusalimschy, L. H. de Figueiredo, W. Celes, “The evolution of Lua,” submitted to ACM HOPL III.
- ▶ N. Ramsey, “Embedding an Interpreted Language Using Higher-Order Functions and Types.” To appear in the *Journal of Functional Programming*, 2005. (A previous version appeared in *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators*.)
- ▶ www.lua.org