Luerl - an implementation of Lua on the Erlang VM
Overview

● Why Erlang
  – The problem
  – The problem domain
  – A bit of philosophy
  – Properties of Erlang

● The Luerl goal

● The result

● The implementation

● The demo

● The comparison
The problem

- Ericsson’s “best seller” AXE telephone exchanges (switches) required large effort to develop and maintain software.

- The problem to solve was how to make programming these types of applications easier, but keeping the same characteristics.
Problem domain

- Lightweight, massive concurrency
- Fault-tolerance must be provided
- Timing constraints
- Continuous operation for a long time
- Continuous maintenance/evolution of the system
- Distributed systems
Some reflections

We were **NOT** trying to implement a functional language

We were **NOT** trying to implement the actor model

**WE WERE TRYING TO SOLVE THE PROBLEM!**
Some reflections

- This made the development of the language/system very focused
- We had a clear set of criteria for what should go into the language/system
  - Was it useful?
  - Did it or did it not help build systems?

The language/system evolved to solve the problem
Properties of the Erlang system

- Lightweight, massive concurrency
- Asynchronous communication
- Process isolation
- Error handling
- Continuous evolution of the system
- Soft real-time
- Support for introspection and monitoring

These we seldom have to directly worry about in a language, except for receiving messages.
Properties of the Erlang system

- Immutable data
- Pattern matching
- Functional language
- Predefined set of data types
- Modules
- No global data

These are what we mainly “see” directly in our languages
The Luerl goal

● A proper implementation of the Lua language
  – It should look and behave the same as Lua
  – It should include the standard libraries

● Should interface well with Erlang
The result

● Implements all of Lua 5.2
  – except goto, _ENV and coroutines
● Seems to manage all tests which don’t use debug
● Interacts well with Erlang
  – Easy for Erlang to call Lua and Lua to call Erlang
  – Compatible with Erlang concurrency and error handling
● Lua’s code handling does not conform to Erlang’s
  – You need to be careful when reloading Lua modules which may reload Erlang modules
The result: Libraries

- **Implemented**
  - Basic Functions
  - Modules (not C-code)
  - String Manipulation
  - Table Manipulation
  - Mathematical functions
  - Bitwise Operations
  - Input and Output Facilities (very few functions)
  - Operating System Facilities (not all functions)

- **Not implemented**
  - The Debug library (too implementation dependant)
The result: Erlang program interface

- Extensive set of functions to call Lua from Erlang
  - Extendable when required
- Straight-forward to call Erlang from Lua
  - No C-interface
The implementation: Lua syntax

- Lua grammar simple, almost LALR(1)
- Can use existing standard Erlang parse–tools
  - Leex for generating tokeniser
  - Yecc for generating parser
    - One reduce–reduce conflict which was easy to handle
The implementation: VM and compiler

- A relatively straight-forward VM
  - Similar, but not the same, as the standard one

- Compiler optimises the environment handling
  - Separates purely local environment of blocks/functions from global environment

- A lot of “unnecessary” information compiled away
  - Error messages very “basic” 😞
The implementation: datatypes

<table>
<thead>
<tr>
<th>Lua</th>
<th>Erlang</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>atom nil</td>
</tr>
<tr>
<td>booleans</td>
<td>atoms true/false</td>
</tr>
<tr>
<td>numbers</td>
<td>floats</td>
</tr>
<tr>
<td>strings</td>
<td>binaries</td>
</tr>
<tr>
<td>tables</td>
<td>array+dict</td>
</tr>
</tbody>
</table>
The implementation: Lua state

● Main difficulty of the implementation
  – Need to implement mutable global data with immutable local data

● We keep all Lua state in one data structure explicitly threaded through everything
The implementation: Lua state

● One big data structure
  – global table store
  – global frame store
  – environment frames
  – tables
  – current stack

● We need to implement our own garbage collector on top of Erlang’s collector for Lua state
The implementation: Lua global data

-record(luerl, {ttab, tfree, tnext, %Table table, free, next
    ftab, ffree, fnext, %Frame table, free, next
    g, %Global table
    stk=[], %Current stack
    meta=[], %Data type metatables
    tag
    }).

-record(meta, {nil=nil, %Unique tag
    boolean=nil,
    number=nil,
    string=nil}).

-record(tref, {i}). %Table reference, index
-record(table, {a, t=[]}, m=nil}). %Table type, array, tab, meta
-record(fref, {i}). %Frame reference, index
The implementation: Lua table store

get_table_key(#tref{}=Tref, Key, St) when is_number(Key) ->
    case ?IS_INTEGER(Key, I) of
        true when I >= 1 -> get_table_int_key(Tref, Key, I, St);
        _NegFalse -> get_table_key_key(Tref, Key, St)
    end;
get_table_key(#tref{}=Tref, Key, St) ->
    get_table_key_key(Tref, Key, St);
get_table_key(Tab, Key, St) ->                  %Just find the metamethod
    case getmetamethod(Tab, <<"__index">>, St) of
        nil -> lua_error({illegal_index,Tab,Key});
        Meth when element(1, Meth) =:= function ->
            {Vs,St1} = functioncall(Meth, [Tab,Key], St),
            {first_value(Vs),St1};              %Only one value
        Meth ->                                 %Recurse down the metatable
            get_table_key(Meth, Key, St)
    end.
The implementation: Lua table store

get_table_key_key(#tref{i=N}=T, Key, #luerl{tabs=Ts}=St) ->
  #table{t=Tab,m=Meta} = ?GET_TABLE(N, Ts),  %Get the table.
  case tt
dict:find(Key, Tab) of
    {ok,Val} -> {Val,St};
    error ->
      %% Key not present so try metamethod
      get_table_metamethod(T, Meta, Key, Ts, St)
  end.

get_table_int_key(#tref{i=N}=T, Key, I, #luerl{tabs=Ts}=St) ->
  #table{a=A,m=Meta} = ?GET_TABLE(N, Ts),  %Get the table.
  case array:get(I, A) of
    nil ->
      %% Key not present so try metamethod
      get_table_metamethod(T, Meta, Key, Ts, St);
    Val -> {Val,St}
  end.
The implementation: Lua table store

def get_table_metamethod(T, Meta, Key, Ts, St) ->
    case getmetamethod_tab(Meta, <<"__index">>, Ts) of
        nil -> {nil,St};
        Meth when element(1, Meth) =:= function ->
            {Vs,St1} = functioncall(Meth, [T,Key], St),
            {first_value(Vs),St1}; %Only one value
        Meth -> %Recurse down the metatable
            get_table_key(Meth, Key, St)
    end.
The implementation: Lua table store

```
set_table_key_key(#tref{i=N}, Key, Val, #luerl{tabs=Ts0}=St) ->
    #table{t=Tab0,m=Meta}=T = ?GET_TABLE(N, Ts0), %Get the table
    case ttdict:find(Key, Tab0) of
        {ok,_} -> %Key exists
            Tab1 = if Val =:= nil -> ttdict:erase(Key, Tab0);
                     true -> ttdict:store(Key, Val, Tab0)
                end,
        ! ->
    Ts1 = ?SET_TABLE(N, T#{t=Tab1}, Ts0),
    St#{luerl{tabs=Ts1}};
```
The implementation: Lua table store

```erlang
error ->
    case getmetamethod_tab(Meta, <<"__newindex">>, Ts0) of
        nil ->
            %% Only add non-nil value.
            Tab1 = if Val =:= nil -> Tab0;
                    true -> ttdict:store(Key, Val, Tab0)
                end,
            Ts1 = ?SET_TABLE(N, T#table{t=Tab1}, Ts0),
            St#luerl{tabs=Ts1};
        Meth when element(1, Meth) =:= function ->
            functioncall(Meth, [Key, Val], St);
        Meth -> set_table_key(Meth, Key, Val, St)
            end
    end.
end.
```
The demo

- Concurrent space ships
  - Logic in Lua
  - Each ship an Erlang process
  - Communicate using Erlang messages
The demo: code

- The default tick move
- The bounce
- The attack tick move
- The zap
- The left/right sectors
The demo: code

```lua
local function move(x, y, dx, dy)
    local nx,ny,ndx,ndy = move_xy_bounce(x, y, dx, dy,
                                            universe.valid_x, universe.valid_y)

    -- Where we were and where we are now.
    local osx,osy = universe.sector(x, y)
    local nsx,nsy = universe.sector(nx, ny)
    if (osx ~= nsx or osy ~= nsy) then
        -- In new sector, move us to the right sector
        universe.rem_sector(x, y)
        universe.add_sector(nx, ny)
        -- and draw us
        esdl_server.set_ship(type, colour, nx, ny)
    end
    return nx,ny,ndx,ndy
end
```
The demo: code

local function move_xy_bounce(x, y, dx, dy, valid_x, valid_y)
    local nx = x + dx
    local ny = y + dy

    if (not valid_x(nx)) then -- Bounce off the edge
        nx = x - dx
        dx = -dx
    end
    if (not valid_y(ny)) then -- Bounce off the edge
        ny = y - dy
        dy = -dy
    end
    return nx, ny, dx, dy
end
The demo: code

```lua
local function move(x, y, dx, dy)
    local nx, ny, ndx, ndy = move_xy_bounce(x, y, dx, dy,
        universe.valid_x, universe.valid_y)

    -- Where we were and where we are now.
    local osx, osy = universe.sector(x, y)
    local nsx, nsy = universe.sector(nx, ny)
    if (osx ~= nsx or osy ~= nsy) then
        -- Zap a nearby ships, only zap when we move
        zap_ships(osx, osy, nsx, nsy)
        -- In new sector, move us to the right sector
        universe.rem_sector(x, y)
        universe.add_sector(nx, ny)
        -- and draw us
        esdl_server.set_ship(style, colour, nx, ny)
    end

    return nx, ny, ndx, ndy
end
```
The demo: code

```lua
local function zap_ships(osx, osy, nsx, nsy)
    local lsx,lsy,rsx,rsy = move_lr_sectors(osx, osy, nsx, nsy)
    local f = universe.get_sector(nsx, nsy)
    if (f and f ~= me) then  -- Always zap ship in front
        ship.zap(f)
    end
    f = universe.get_sector(lsx, lsy) or
        universe.get_sector(rsx, rsy)
    if (f and f ~= me) then  -- Zap ship either left or right
        ship.zap(f)
    end
end
```
The demo: code

```lua
local function move_lr_sectors(osx, osy, nsx, nsy)
    local idx, idy = nsx-osx, nsy-osy
    local lsx, lsy, rsx, rsy -- Left, right of next sectors
    if (idx == 0) then
        lsx, lsy = nsx - idy, nsy
        rsx, rsy = nsx + idy, nsy
    elseif (idy == 0) then
        lsx, lsy = nsx, nsy - idx
        rsx, rsy = nsx, nsy + idx
    elseif (idx == idy) then
        lsx, lsy = nsx - idx, nsy
        rsx, rsy = nsx, nsy - idy
    else
        -- idx ~= idy
        lsx, lsy = nsx, nsy - idx
        rsx, rsy = nsx - idx, nsy
    end
    return lsx, lsy, rsx, rsy
end
```
Alternatives

● External Lua system
  – Through Erlang “ports” to other OS processes

● Include Lua engine inside Erlang
  – Using Erlang NIFs to call Lua engine
Which one?: Lua in Erlang

+ Complete access to Erlang/VM properties
+ Easier use of Erlang concurrency
+ Faster interface
+ Only need one system

– Slower
– Data sharing difficult
Which one?: external Lua system

+ Faster Lua
+ Probably able to run more code

– Generally slower interface
– More difficult to use Erlang concurrency
– More difficult to get parallelism
Thank you

Robert Virding: rvirding@gmail.com
@rvirding