Luerl documentation

Luerl is an implementation of standard Lua 5.2 written in Erlang/OTP.

Contents

Preface

Lua is a powerful, efficient, lightweight, embeddable scripting language common in games, IoT devices, AI bots, machine learning and scientific computing research.

It supports procedural, object-oriented, functional, data-driven, reactive, organizational programming and data description.

Being an extension language, Lua has no notion of a "main" program: it works as a library embedded in a host. The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and call Erlang functions by Lua code.

Through the use of Erlang functions, Luerl can be augmented to cope with a wide range of different domains, creating a customized language sharing a syntactical framework.

Luerl is a library, written in clean Erlang/OTP. For more information, click on the get started tutorial. You may also browse the examples and learn from the luerl_demo source code.

Data representation

Luerl represents data in two ways, an internal and external formats. The external format is designed to interface with native Erlang while the internal format represents data as it is handled inside this implementation of the Lua virtual machine and interpreter.

The luerl library has two groups of functions, those ending in **1** and those that don't. Those that don't assume that the arguments are in external format and automatically encodes them, while those ending in **1** assume they are in internal format.

In most cases using the external format makes code more readable and understandable. However saving the translation costs by directly manipulating data in the internal format can be worth it, once a function for the internal representation is used you are hooked and there is no going back!

To illustrate the following two code examples are identical the first using the external representation the second using the internal one:

```
St0 = luerl:init().
F = fun(_, StIn) ->
    {[[{<<"a">>, 1.0}]], StIn}
  end.
St1 = luerl:set_table([t], F, St0).
luerl:eval("return t()", St1).
```

```
S0 = luerl:init().
F = fun ([], Sti) ->
    {T, Sto} = luerl_emul:alloc_table([{<<"a">>, 1.0}], Sti),
    {[T], Sto}
  end.
S1 = luerl:set_table1([<<"t">>], {function, F}, S0).
luerl:eval("return t()", S1).
```

Data types

For the implementation of Lua data types we internally use their corresponding Erlang type:

nil - atom nil

true/false - atoms true/false

strings - string binaries

numbers - floats

tables - #table{} with array for keys 1..n, ordict for rest

functions - #function{} or {function,Fun}

See luerl.hrl for the field names of the records.

All tables are combinations of Erlang ttdicts and arrays. In each table an array is used for integer keys while an ttdict is used for all other keys.

We use this information when building and processing tables.

Nil

Nil is a type represented by the Erlang atom **nil**, whose main property is to be different from any other value. Lua uses **nil** as a kind of non-value, to represent the absence of a useful value.

Booleans

Erlang atoms **true** and **false**.

Strings

Strings are represented the same in both formats, luerl uses binary strings list strings will be interpreted as a table!

Numbers

Numbers are the same in both internal and external representations, they are simple float numbers. Please be aware that in Lua < 5.3 numbers are always treated as floats.

Tables

Tables in Lua are known as associative arrays. All Lua types can be used as keys, except **nil**.

Tables are the only data structure mechanism in Lua, tables implement many data types in simple and efficient ways, lists, sets, arrays, sparse matrices, structures, the kitchen sink, you name it. Tables in Lua are also used for several other purposes like global variables, modules, object and classes, next are some notes about the insides of its Luerl implementation.

In the external format tables are represented as lists of tuples, where the first element is the index and the second the value or as lists of values where the index is implicitly assigned by their position in the list (starting with 1). The tables [1] and [{1,1}] are identical.

In the internal format tables are represented as references, to allocate a table reference the function luerl_emul:alloc_table/2 is that returns a new reference and a state that includes this reference then given a name using luerl:set_table1/3. Single keys can be accessed using luerl:get_table1/2 to resolve a name to a reference and then luerl_emul:get_table_keys/3 to get the value for a given key in the table.

Functions

In the external representations functions are simply functions of the **arity/2**, where the first argument is a list of function arguments and the state at the time of function execution.

They return a tuple with a list of return values as the first argument and the new state.

All parameters are in the external format and all return values need to be in the external format.

In the internal format functions are a tuple with the first element being the atom **function** and the second a function of the **arity/2**.

Both the return values as well as the function parameters are in the internal format as well.

Getting started

Lua is implemented as a library to start first include luerl on your OTP application.

For this tutorial we're going to use erlang.mk to setup quickly our environment, build our code and start or stop the project release.

Erlang.mk is very easy to setup: create a folder, put Erlang.mk in it, and write a Makefile:

include erlang.mk

For a step by step:

$ mkdir luerl_demo

$ cd luerl_demo

$ curl https://erlang.mk/erlang.mk -o erlang.mk

Create a Makefile and copy the next code snippet, there you can make the correct changes to reflect your project needs.

```
PROJECT = ophelia
DEPS = luerl
include erlang.mk
```

$ make

From that point you can create an src/ folder or start using http://erlang.mk/ templates - User guide.

Hello world

$ git clone https://github.com/nonsensews/luerl_demo.git

$ cd luerl_demo

$ curl https://erlang.mk/erlang.mk -o erlang.mk

$ make

$ ./hello.sh

Show me the ships!

You first need to get sdl2 and sdl_image installed on your machine then you'll be all set.

$ cd ships

$ ./demo.sh

Hello Erlang

The Lua interpreter runs entirely inside its own Virtual Machine implemented in Erlang/OTP, however by itself has no access to any Erlang functions or modules.

This means that the only way to extend the reach of a Lua script is by filling the tables in the interpreter state with Lua functions that can reach over that boundary.

On the most basic level the standard libraries (luerl_lib_...) provide some of those functions. When luerl:init() is run to create a new Lua state those libraries get loaded and their functions added to the related tables, luerl_lib_io is loaded to io and so on.

That is a good start for basic programs but in the case we use luerl to extend our own program with scripting capabilities we need more. Adding our own functions (that call our Erlang logic) can be done by using luerl:set_table/3 with a function as the second argument.

Functions

A few words should be said about how functions are handled in Luerl. Here a Lua function maps to an Erlang function with two arguments:

- the list of arguments passed to the Lua function
- the current state of the Lua interpreter

It then does it's work and return a tuple of two arguments:

- a list of return values
- the new state of the Lua interpreter

So the call:

add(1, 2) # => 3

would look like this for the Erlang code interpreting it:

```
add([1, 2], St) ->
  {[1 + 2], St}
```

Return values

We can return a few value types from a function (as each element of the return list):

- a number, always be treated as a float
- a binary string
- a table, a list of tuples where the first argument is a key and the second is its value.
- a list (which will then be translated into a table with integer indexes)
- a function (of **arity**/**2**)

Examples

Add function

A very trivial example would be a add function:

```
%% Initiate an empty state.
St0 = luerl:init().
%% Put the add function in the main table of the state (and get a new one)
St1 = luerl:set_table([add], fun([A, B], St) -> {[A + B], St} end, St0).
%% evaluate a little example script.
luerl:eval("return add(1, 1)", St1).
```

A more complex example would be creating a process that counts things (yes that can also be done with variables but let's ignore that part). We implement a function called counter that will spawn a new erlang process that waits for messages. Then we return a table with 3 functions (add, sub, and get) that act as closures around the processes PID. We now have a lua table that can uses send, receive and spawn all functions that usually can't be accessed from luerl directly.

Counter process

```
%% Our loop function for the process
Loop = fun(N, L) ->
       receive
         {get, Pid, Ref} ->
           Pid ! {count, Ref, N},
           L(N, L);
         {add, X} ->
           L(N + X, L);
         {sub, X} ->
           L(N - X, L)
        end
       end.
%% The initial state.
St0 = luerl:init().
%% We add the counter function that returns a table
St1 = luerl:set_table([counter], fun([Initial], St) ->
%% Spawn a process with the loop function (defined above).
Pid = spawn(fun() -> Loop(Initial, Loop) end),
%% Create a table (a list of tuples) to return
%% those anonymous functions use Pid which is defined in
%% the current scope but won't be visible outside or to lua.
R = [{add, fun([N], StF) -> Pid ! {add, N}, {[N], StF} end},
    {sub, fun([N], StF) -> Pid ! {sub, N}, {[N], StF} end},
    {get, fun([], StF) ->
           Ref = erlang:make_ref(),
           Pid ! {get, self(), Ref},
           receive
             {count, Ref, N} ->
               {[N], StF}
           after
             10000 ->
               {[timeout], StF}
           end
         end}],
%% return the table we just created
{[R], St}
end, St0).
%% We call a little script now it will:
```

```
%% 1) create a new counter
%% 2) print the initial state
%% 3) add 1 to the counter
%% 4) print the changed state
luerl:eval("c = counter(0); print(\"0: \", c.get()); c.add(1); print(\"1: \", c.get())", St1).
%% The output will be:
%%   1: 0
%%   2: 1
```

Hello examples

To run the examples, do make and then start the Erlang command line with erl -pa ./ebin.
Don't be shocked by the very long dump following each function call.
At the command line you are seeing the Lua State dumped, that is returned by these calls:

execute a string

luerl:do("print(\"Hello, Robert!\")").

execute a file

luerl:dofile("./examples/hello/hello.lua").

separately parse, then execute

EmptyState = luerl:init(),
{ok, Chunk, State} = luerl:load("print(\"Hello, Chunk!\")", EmptyState),
{_Ret, _NewState} = luerl:do(Chunk, State).

call a function in the state

{Res,State1} = luerl:call_function([table,pack], [<<"a">>,<<"b">>,42], State0)

executes the call table.pack("a", "b", 42) in State0. E.g.:

{Res,State1} = luerl:call([table,pack], [<<"a">>,<<"b">>,42]),
io:format("~p~n",[Res]).

call a method in the state

{Res,State1} = luerl:call_method([g,h,i], [<<"a">>,<<"b">>,42], State0)

executes the call g.h:i("a", "b", 42) in State0.

define a method in the state

State1 = luerl:set_table([inc], fun([Val], State) -> {[Val+1], State} end)

the method can be called like this:

luerl:do(<<"print(inc(4))">>, State1)

Luerl examples

For more information see the examples directory:

./hello.erl is very brief while examples/hello/hello2.erl offers a comprehensive list on how use the individual interface functions.

You can build and run these samples with:

make examples

Hello Lua

First you need to define an interface module written in Erlang with the functions you want to call from the standard Lua 5.2 implemented by Luerl.

There is a predefined way of building and loading such a modules so it functions can be reached from Lua through a table like any other Lua function.

There is also a predefined way of passing data in and out of these functions. Take a look at the source code all the modules *luerl_lib_XXXX.erl* are interface modules like this. If you check in the module *luerl_emul.erl* in the function **load_lib/3** you can see how it is installed.

One reason for having it this way is that it makes sandboxing very easy as Lua code cannot do this loading, it has to be done in the surrounding Erlang environment.

LUA_PATH

?

Sandboxing

Sometimes we want to do exactly the opposite, not add functionality to what the luerl interpreter can do but remove it. Since, as discussed above, the only way it can access functionality is if its stored in one of the tables (that reside in the state) restricting functionality is as simple as removing some of the tables.

If we for example we really hate math and want to make sure no mathematical functions are ever called in scripts we run we could do the following:

```
St0 = luerl:init().
St1 = luerl:set_table([math], <<"no math for you!">>, St0).
luerl:eval("return math", St1).
```

Of cause this also works with other tables, like **io** or **file** which would prevent a Lua script to access files or perform IO-operations (not as bad as forbidding math but still helpful).

Installing Luerl

Luerl implement standard Lua 5.2 in pure Erlang to start using the project you need to include the luerl library as new dependency on your OTP application.

Hex

You can install luerl from hex.pm by including the following in your rebar.config:

```
{deps,[
   {luerl, "X.Y.Z"}
]}.
```

where *X.Y.Z* is one of the release versions.

For more info on rebar3 dependencies see the rebar3 docs.

Erlang.mk

Or use erlang.mk to setup quickly your environment, build the code and project release.

Create a Makefile and include luerl to the project dependencies.

```
DEPS = luerl
```

Source code

```
$ git clone https://github.com/rvirding/luerl.git
```

Interface functions

All functions optionally accept a **Lua State** parameter. The Lua State is the state of a Lua VM instance. It can be carried from one call to the next. If no State is passed in, a new state is initiated for the function call.

As it is possible in Lua to create self-referencing data structures, indeed the standard libraries have many instances of this, then using the functions which decode their return values can cause an infinite loop during the decoding. An simple example is the top level table which contains a key **_G** which references the top-level table.

Note that Lua **Chunks** (see definition below) can travel between different States. They are precompiled bits of code, independent of State. That you can 'carry around' this is no unique to Luerl but a low-level implementation detail of the standard Lua language, for more on chunks read the official Lua 5.2 reference manual.

Spec Definitions

Binary means an Erlang binary string.
Chunks means a portion of precompiled bytecode.
State means a Lua State, this *is* a Lua VM instance.
Path means a file system path and file name.
KeyPath means an Erlang list of **atoms** representing nested names, e.g. [table,pack] for table.pack.
Keys means Lua table keys, the keys of a key-value structure.

Hello Examples

See below and files hello.erl and hello2.erl in examples/hello/.

Functions

eval and **do** functions differ only in what they return. The **do** functions return results and a new Lua State, the **eval** functions return a tuple starting on 'ok' or 'error', then the result, or cause of error.

do --> {Result, State}

eval --> {ok, Result} | {error, Reason}

luerl:eval(String|Binary|Form[, State]) -> {ok, Result} | {error, Reason}.

Evaluate a Lua expression passed in as a string or binary, and return its result.

luerl:evalfile(Path[, State]) -> {ok, Result} | {error, Reason}.

Load and execute a file, and return the result.

luerl:do(String|Binary|Form[, State]) -> {Result, NewState}.

Evaluate a Lua expression and return its result, and the new Lua State.

luerl:dofile(Path[, State]) -> {Result, NewState}.

Load and execute the Lua code in the file and return its result, and the new Lua State. Equivalent to doing

luerl:do("return dofile('FileName')").

luerl:load(String|Binary[, State]) -> {ok,Function,NewState} | {error, Reason} .

Parse a Lua chunk as string or binary, and return a compiled chunk ('form').

luerl:loadfile(FileName[, State]) -> {ok,Function,NewState}.

Parse a Lua file, and return a compiled chunk ('form').

luerl:path_loadfile([Path,], FileName, State) -> {ok,Function,FullName,State}.

Search Path until the file FileName is found. Parse the file and return a compiled chunk ('form'). If Path is not given then the path defined in the environment variable LUA_LOAD_PATH is used.

luerl:load_module(KeyPath, ErlangModule, State) -> State.

Load ErlangModule and install its table at KeyPath.

luerl:load_module1(KeyPath, ErlangModule, State) -> State.

Load ErlangModule and install its table at KeyPath.

luerl:init() -> State.

Get a new Lua State = a fresh Lua VM instance.

luerl:call(Form, Args[, State]) -> {Result,State}

luerl:call_chunk(Form, Args[, State]) -> {Result,State}

Call a compiled chunk or function. Use the call_chunk, call has been kept for backwards compatibility.

luerl:call_function(KeyPath, Args[, State]) -> {Result,NewState}

Call a function already defined in the state. KeyPath is a list of names to the function. KeyPath, Args and Result are automatically encode/decoded.

luerl:call_function1(KeyPath, Args, State) -> {Result,NewState}

Call a function already defined in the state. KeyPath is a list of keys to the function. KeyPath, Args and Result are **NOT** encode/decoded.

luerl:call_method(MethPath, Args[, State]) -> {Result,NewState}.

Call a method already defined in the state. MethPath is a list of names to the method. MethPath, Args and Result are automatically encode/decoded.

luerl:call_method1(MethPath, Args, State) -> {Result,NewState}

Call a method already defined in the state. MethPath is a list of keys to the method. Keys, Args and Result are **NOT** encode/decoded.

luerl:stop(State) -> GCedState.

Garbage collects the state and (todo:) does away with it.

luerl:gc(State) -> State.

Runs the (experimental) garbage collector on a state and returns the new state.

luerl:set_table(Path, Value, State) -> State.

Sets a value inside the Lua state. Value is automatically encoded.

You can use this function to expose an function to the Lua code by
using this interface:
fun(Args, State) -> {Results, State}
Args and Results must be a list of Luerl compatible erlang values.

Known bugs

Some things which are known not to be implemented or work properly:

- coroutines and goto
- only limited standard libraries
- proper handling of _ENV
- tail-call optimization in return

Functions defined in a loop, *while*, *repeat* and *for*, **and** when the loop is exited with a *break* from inside an *if* will generate an error when called. For example the functions defined in

```
for i=1,10 do
  a[i] = {set = function(x) i=x end, get = function () return i end}
  if i == 3 then break end
end
```

Note: This only occurs if the loop is actually exited with the break, otherwise there is no problem.

Standard library

There is also a library module, luerl_lib, which contains functions which may be used.

luerl_lib:first_value(ReturnList) -> Value.

luerl_lib:is_true_value(ReturnList) -> true | false.

Current implementation functions

- _G
- _VERSION
- assert
- collectgarbage
- dofile
- eprint
- error
- getmetatable
- ipairs
- load
- loadfile
- next
- pairs
- pcall
- print
- rawequal
- rawget
- rawlen
- rawset
- require
- select
- setmetatable
- tonumber
- tostring
- type
- unpack
- io.flush
- io.write
- math.abs
- math.acos
- math.asin
- math.atan
- math.atan2
- math.ceil
- math.cos
- math.cosh
- math.deg
- math.exp
- math.floor
- math.fmod
- math.frexp
- math.huge
- math.ldexp
- math.log

- math.log10
- math.max
- math.min
- math.modf
- math.pi
- math.pow
- math.rad
- math.random
- math.randomseed
- math.sin
- math.sinh
- math.sqrt
- math.tan
- math.tanh
- os.clock
- os.date
- os.difftime
- os.getenv
- os.time
- package.config
- package.loaded
- package.preload
- package.path
- package.searchers
- package.searchpath
- string.byte
- string.char
- string.find
- string.format (should handle most things now)
- string.gmatch
- string.gsub
- string.len
- string.lower
- string.match
- string.rep
- string.reverse
- string.sub
- string.upper
- table.concat
- table.insert
- table.pack
- table.remove
- table.sort
- table.unpack
- bit32.band
- bit32.bnot
- bit32.bor
- bit32.btest
- bit32.bxor
- bit32.lshift
- bit32.rshift
- bit32.arshift
- bit32.lrotate
- bit32.rrotate
- bit32.extract

- bit32.replace
- debug.getmetatable
- debug.getuservalue
- debug.setmetatable
- debug.setuservalue

The Lua API

Lua is an embeddable language implemented as a library that offers a clear API for applications inside a register-based virtual machine.

This ability to be used as a library to extend an application is what makes Lua an extension language.

At the same time, a program that uses Lua can register new functions in the Lua environment; such functions are implemented in Erlang (or another language) and can add facilities that cannot be written directly in Lua. This is what makes Lua an extensible language.

These two views of Lua (as extension language and as extensible language) correspond to two kinds of interaction between Erlang and Lua. In the first kind, Erlang has the control and Lua is the library. The Erlang code in this kind of interaction is what we call application code.

In the second kind, Lua has the control and Erlang is the library. Here, the Erlang code is called library code. Both application code and library code use the same API to communicate with Lua, the so called Luerl API.

Modules, Object Oriented programming and iterators need no extra features in the Lua API. They are all done with standard mechanisms for tables and first-class functions with lexical scope.

Exception handling and code load go the opposite way: primitives in the API are exported to Lua from the base system C, JIT, BEAM.

Lua implementations are based on the idea of closures, a closure represents the code of a function plus the environment where the function was defined.

Like with tables, Lua itself uses functions for several important constructs in the language.

The use of constructors based on functions helps to make the API simple and general.

There are no coroutines in Luerl it may seems counter intuitive coming from a more common Lua background.

In this ecosystem you always want to use processes instead, the BEAM Virtual Machine it's build for handling independent isolated processes that are very small and almost free at creation time and context switching. The main difference between processes and coroutines is that, in a multiprocessor machine a OTP release on the BEAM Virtual Machine runs several processes concurrently in parallel.

Coroutines, on the other hand, runs only one at the time on a single core and this running coroutine only suspends its execution when it explicitly requests to be suspended.

Lua Virtual Machine

The Lua VM is a hybrid implementation. It uses normal Erlang function calls for Luerl calls and blocks and has a small instruction set for operations inside a block. This is a pure stack machine.

Blocks keep variables in tuples. There are two variable types depending on how they are defined:

- Local variables that are used in this block and sub-blocks, but not used in any functions defined in the blocks. These are kept in a stack of tuples, the LocalVars or Lvs, and referenced by offset in stack and offset in tuple.
- Environment variables that are defined in functions which are defined in this block or in sub-blocks. This mean they must be kept around as long as the functions are alive and are stored in the global heap as each invocation can modify them. They are kept in a stack of references, the EnvironmentVars or Evs, to tuples in the global heap and referenced by offset in stack and offset in tuple.

A function contains a reference to the stack of environment variables which existed when it was created. Note that the mutable nature of Lua data means that these can be modified and the changes must be visible to every function which references them.

There is also a stack containing arguments and temporary values. This is stack is "global" in the sense that it is passed through all calls and blocks. It is also passed as an argument into functions implemented in Erlang. This is so that event of a Lua/Luerl GC the collector uses the stack to determine which data in the global heap is to be saved.

To handle multiple return values we always return a list of values. The only place this is not done is in luerl_eval.erl when getting values from the environment where we can only have one value. This means a lot of calls to first_value/1 in luerl_emul.erl, but the consistency is worth it.

Similarily all the arguments in a function call are passed in a list. The function then unpacks the list into its arguments, including '...'.

All of the predefined libraries have an install/1 function. This is called when initialising Luerl; it does any library specific initialisation necessary and returns a table containing the functions in the library.

We create a unique tag which is saved in the environment. This is used so we can implement 'break' with a simple throw. The thrown value includes the tag so we can uniquely catch it and not get confused with a throw/error/exit from the erlang code.

Garbage collector

Is important to note that the garbage collector is never called by luerl itself.

The main reason is that we could not work out a decent heuristic about when to call it. For example there is no heap as such we can check to see how much luerl memory we are using and when it fills. So it is left up to the caller to do so. Currently calling it from inside luerl is a not an option so you need to call it from the Erlang level.

This might be useful if you want to reuse the initial state for each call as you can just throw away the returned state and not waste time doing garbage collection.

Compiler

The compiler has state at different levels:

- In luerl_comp there is #comp{} containing code, options and errors.
- In the #cst{} between the compiler modules for data outside the code. This empty so far.
- Inside and local to the compiler modules.

All the compiler modules are written so that they chain a status argument through their code, even if it not used. When they are not used we just send the atom 'nil' through and check it comes out "the other end".

When to use Luerl

Fast Language Switch: Luerl should allow you to switch between Erlang and Lua incredibly fast, introducing a way to use very small bits of logic programmed in Lua, inside an Erlang application, with good performance.

Multicore: Luerl provides a way to transparently utilize multicores. The underlying Erlang VM takes care of the distribution.

Microprocesses: It should give you a Lua environment that allows you to effortlessly run tens of thousands of Lua processes in parallel, leveraging the famed microprocesses implementation of the Erlang VM. The empty Luerl State footprint will be yet smaller than the C Lua State footprint.

Forking Up: Because of the immutable nature of the Luerl VM, it becomes a natural operation to use the same Lua State as a starting point for multiple parallel calculations.

However, Luerl will generally run slower than a reasonable native Lua implementation. This is mainly due the emulation of mutable data on top of an immutable world. There is really no way around this. An alternative would be to implement a special Lua memory outside of the normal Erlang, but this would defeat the purpose of Luerl. It would instead be then more logical to connect to a native Lua.

Some valid use cases for Luerl are:

- Lua code will be run only occasionally and it wouldn't be worth managing an extra language implementation in the application.
- The Lua code chunks are small so the slower speed is weighed up by Luerl's faster interface.
- The Lua code calculates and reads variables more than changing them.
- The same Lua State is repeatedly used to 'fork up' as a basis for massively many parallel calculations, based on the same state.
- It is easy to run multiple instances of Luerl which could better utilise multicores.

There may be others.