Lua errors in the face of multiple VMs

Lua offers exceptions as a form of error handling for programs. However it has no support for user defined types that could be used to match the error received. This lack has profound differences on how Lua exceptions are used as compared to other languages. The only chance of having a structural error object is by using Lua tables. This table protocol alone still suffers from the same deficiency when it comes to matching the error type/category. In this blog post, I'll explore the solution adopted in Emilua and further challenges.

Several properties may be analysed when designing error propagation mechanisms. Properties commonly weighted by exception designers are:

> **Clean code**
>
> The treatment of errors should be in a separate layer of code and as much invisible as possible. So the code reader could notice the presence of exceptional cases without stop his reading.
>
> **Information on errors**
>
> The errors should carry out as most as possible information from their origin, causes and possibly the ways to resolve it.
>
> **Non-Intrusive error**
>
> The errors should not monopolize a communication channel dedicated to the normal code flow. They must be as discrete as possible. For instance, the return of a function is a channel that should not be exclusively reserved for errors.
>
> — A proposal to add a utility class to represent expected monad, https://isocpp.org/files/papers/n4109.pdf (selected excerpts)

Information on errors is the property most influenced by OO designs commonly found on other languages. The presence of OO means you can match on error types in a hierarchical-fashion from the most specific to the most general. That's just not an option for Lua. Therefore different patterns will apply. It's useful to revisit the need for custom information on errors before we talk about solutions for Lua tables.

By using exceptions, you group several code statements into units of code blocks. Any error raised from this block will appear all the same to you. If a failing operation on this code block requires you to grow buffers, but a failure on another operation requires the whole block to be aborted, there's no way you can do the proper decision with just this level of (non-)information. The way out is to extend the error object with context about the error's origin and type. That's exactly what subclassing allows in OO-languages.

Lua tables already satisfy the requirement to hold extra information as needed. You just insert new elements in the table and you're done. The open question here is how to "tag" a table as being part of a specific error type. There are two approaches that I see for this problem.

The first approach is going protocol-centric by outlining a rule where every error object should have a standard field identifying the error type. The code below illustrates this approach.

```lua
local read_error = require('read_error')
local write_error = require('write_error')

local ok, e = pcall(block)
if not ok then
    if e.type == read_error then
        -- ...
    elseif e.type == write_error then
        -- ...
    else
        -- ...
    end
end
```

The second approach is to just rely on Lua metatables for the tagging aspect. The illustration for this alternative approach follows.

```lua
local read_error = require('read_error')
local write_error = require('write_error')

local ok, e = pcall(block)
if not ok then
    if getmetatable(e) == read_error then
        -- ...
    elseif getmetatable(e) == write_error then
        -- ...
    else
        -- ...
    end
end
```

Neither approach can handle the subclassing case from OO to match errors in a hierarchical-fashion, so we just ignore this feature. I can imagine how language lawyers infiltrated in the Lua community will scream their guts out that metatables are the more idiomatic choice here, but the truth is: this is just a minor issue on the whole topic. We'll be focusing on more important matters now.

Neither approach provides an answer on how you should be creating new error categories (e.g. the `read_error` from the examples). However that's pretty much my main concern here. There are two issues that language lawyers would fail to give enough thought:

- Lua is a glue language. Therefore a few errors will have their origin in native code. Native code has no concept of Lua tables.
- The Lua solution to exploit parallelism is to spawn several VMs. A Lua table declared in one VM won't have the same meaning in another VM. Emilua programs can make use of multiple VMs, so that's relevant.

Both issues imply the same restriction on our solution: Lua tables can't be used to represent an error category. The way out of this problem is rather easy fortunately: once you realize error categories are global resources you'll see that reusing the machinery created for modules is an option.

Files are global resources and so are modules. A path uniquely identifies some resource and the underlying resource is guaranteed to be the same across the whole process. So we just hijack the module machinery to also be able to import the "error category" resource. In Lua, the error category will be an userdata wrapping the real category. This design allows the category to have the same meaning across native code and all of the VMs. Now the only question remaining is how to represent native errors.

`errno`

The most fundamental piece of error propagation on UNIX operating systems (and their heirs) is the `errno` variable. Check the documentation for some random syscall and you'll soon stumble on `errno` values that may be set. Emilua strives to offer good async IO support so it always map a low-level interface after some "fibrillation" is done on top. That means we must start on `errno` for our design.

`errno` is just a big enumeration of error codes that the OS can publish. The integer value is implicitly typed on this enumeration. You cannot add new values to this enumeration so we only have one category here and no solution yet. Fortunately other people faced the same issue a long time ago so we can just stand on their shoulders and save ourselves big time. In C++ the solution is `std::error_code` and that's just a 2-tuple value made up of an integer value in the likes of `errno` and an associated error domain. The error domain is the thing we've been after.

So far I've been trying to be didactic and present a rationale all the time. However I feel like I can make a change of tone here. If you're curious about the rationales for `std::error_code`, you can do your own homework. As for me, I think I only need to present how to use the Lua solution as that's what most likely will interest you.

Lua errors in Emilua

Emilua will translate `std::error_code` objects from the native layer to Lua tables. The table has two values:

`code: number`

The integer `errno`-like value.

`cat: userdata`

The error category.

Extra information may be attached to the raised error object now and then. Native code can declare error categories and they can cross the boundaries between VMs without losing any meaning.

That's half of the story. The other half is how to allow Lua code to declare new error categories. And the plan (that's not implemented yet as it touches the module system) is to allow the user to write module files that have a file extension other than `*.lua` (e.g. `*.err.lua`). The contents for such modules will be in the likes of:

```lua
assert(_CONTEXT == 'ecat')
return {
    { "lua_id", "error description" },
    { "not_found", "could not found the requested target" },
    { "eperm", "permission error" }
}
```

The `errno`-like value will be taken from the order in the table. The path to the module will ensure uniqueness among all Lua VMs. This code will be executed in a separate Lua state once and the result will be reused for all other VMs as to avoid leaking side-effects.

The solution achieves a good level of simplicity. However that's also an unusual design (in the Lua land) so I thought this text would be useful to unveil the rationales. I'm publishing it rather early anyway considering that only half of the machinery in Emilua is ready by the moment of this writing.