Titan: A System Programming Language made for Lua

Hugo Musso Gualandi, PUC-Rio
in collaboration with André Maidl, Fabio Mascarenhas,
Gabriel Ligneul and Hisham Muhammad

Part 1: Why Titan

- We started out interested in optimizing compilers and interpreters for Lua.
  - To make our programs run faster
  - So we can write high-level code without feeling guilty about performance (!)

- Different goal from Typed Lua. (See André's talk)

Because if it isn’t fast, we will find another way…

```lua
-- Caching globals
local sfind  = string.find
local smatch = string.match

-- Avoid table.insert
xs[#xs + 1] = blah

-- Avoid ipairs
for i = 1, #xs do
  local x = xs[i]
end
```

Two ways to go fast

1) Optimizing Lua implementation (LuaJIT)
2) Use a different language (via the C API)

1) Optimize Lua

- State of the art: just-in-time compilation
  - Collect run-time information
  - Speculatively specialize and optimize
  - Fall back to interpreter if needed

- Lua is lucky to have LuaJIT, a best-in-class JIT.

JIT problems

- Building a JIT is labor-intensive
  - Fundamentally challenging
  - Tooling is still an open problem
  - (Hard to keep up with language evolution)
- Doesn’t optimize evenly
  - Up to 10x difference between compiled and interpreted code

2) Use a different language

- Perhaps we are trying to use Lua beyond what it was designed for?
- “Code the performance-sensitive parts in C”
- Original idea behind scripting languages

Two languages, playing to their strengths

Scripting Language	System Language
Dynamically Typed	Statically Typed
Interpreted	Compiled
Glue Code	Core Components
Flexible & Expressive	Structured & Efficient

C problems

- C-API is hard to use
  - The one thing never in the Lua tutorials
  - Stack-based
  - Mismatched language semantics
- Only worth it for large chunks of code
  - Rewriting existing code is a lot of work
  - Runtime overhead in language boundary (see various lua-to-C compilers)

Part 2: What is Titan?

Titan is a new **statically-typed** system language, **focused on performance**. It is designed to **seemlessly interoperate** with Lua, and should feel familiar to Lua programmers.

(We are currently working on a proof-of-concept implementation. Could still change significantly)

A Glimpse of Titan

```
function sum_list(xs: {integer}) : integer
  local sum: integer = 0
  for i: integer = 1, #xs do
    sum = sum + xs[i]
  end
  return sum
end
```

Titan is Similar to Lua

```
function sum_list(xs: {integer}) : integer
  local sum: integer = 0
  for i: integer = 1, #xs do
    sum = sum + xs[i]
  end
  return sum
end
```

- Familiar syntax, looks like “Lua with Types”
  - But isn’t Typed Lua – (See André’s talk)
- Semantics is close to a subset of Lua

Titan is Statically Typed

```
function sum_list(xs: {integer}) : integer
  local sum: integer = 0
  for i: integer = 1, #xs do
    sum = sum + xs[i]
  end
  return sum
end
```

- Compiles into efficient code
- Compiler-checked documentation

Titan plays along with Lua

```
function sum_list(xs: {integer}) : integer
  local sum: integer = 0
  for i: integer = 1, #xs do
    sum = sum + xs[i]
  end
  return sum
end
```

- Titan modules can be require-ed from Lua
- Titan can work with Lua datatypes
- Titan shares the Lua garbage collector.
- Calling Titan from Lua (and vice versa) should be very cheap

Performance is a goal: Restrictions

```
function sum_list(xs: {integer}) : integer
  local sum: integer = 0
  for i: integer = 1, #xs do
    sum = sum + xs[i]
  end
  return sum
end
```

- Some things are errors in Titan, which helps us generate efficient code:
  - If `xs` is not a list, throws an error
  - If `xs[i]` is not an integer, throws an error
  - ...

Performance is a goal: New Abstractions

```
struct Point
  x: float
  y: float
end

function mid(p: Point, q: Point): Point
  local x: float = (p.x + q.x) / 2.0
  local y: float = (p.y + q.y) / 2.0
  return Point.new(x, y)
end
```

LuaJIT-style FFI

```
foreign C [[
  double hypot(double, double);
]]

function pythagoras(): float
  return C.hypot(3.0, 4.0)
end
```

- Easy feature to add to a typed language
- Convenient way to create bindings
- Automatically converts inputs and outputs
- No C-API overhead (for Titan callers)

Part 3: How to implement?

- How to be interoperable with Lua?
  - How do we expose Titan code to Lua?
  - How does Lua’s GC collect Titan’s garbage?
- How to be efficient?
  - Choices in language semantics
  - How do we generate code?
  - How do we avoid C-API overhead?

Exposing Titan code

- We compile Titan modules to an “so” file (similar to a C module)
- Exported Titan functions use the C-API calling convention (receive a lua_State*, etc)
- From Lua’s point of view,
  calling Titan is like calling C

Sharing the GC

- Common issue when mixing two languages
- We aim to use Lua's GC without modifications

- Titan datatypes
  - Implemented as Lua arrays (not userdata)
  - Similar to Python's namedtuples
- Titan functions (local variables)
  - Primitive values saved on C stack
  - GC objects saved on Lua stack as well

Being optimization-friendly

- Static typing
    - More efficient primitive values
    - Cheaper function calls
- Fail early
    - Avoid expensive fallback paths
- Optimization-friendly data types
    - structs instead of hash tables
    - C types for FFI

Code generation

- Compile to native code
  - No interpreter overhead
- Reuse existing tooling
  - Lots of options for compiling typed languages (GCC, Clang, LLVM, ...)
- Currently an AOT compiler targeting C (to keep things simple)

Bypassing the C-API

- The C-API is “dynamically typed”
  - Operations can receive any Lua object
  - Lots of error checking
  - Programmer convenience (stack ajusting)
- Titan accesses the guts of the interpreter.
  - Measurably faster, allows more specialization
  - (Tradeoff is implementation challenge and tying each Titan version to a minor version of Lua)

Example: Array write

```
xs[i] = 17
```

```c
#include "ltable.h"
#include "lvm.h"

{
    Table *t = local_xs;
    lua_Integer k = local_i;
    int v = 17;
    const TValue *vt = L->ci->func + 2;

    unsigned int actual_i = l_castS2U(k) - 1;
    unsigned int asize = t->sizearray;

    if (actual_i < asize) {
        TValue *slot = &t->array[actual_i];
        setivalue(slot, v);
    } else {
        TValue *slot = (TValue *) luaH_getint(t, k);
        TValue vk; setivalue(&vk, k);
        TValue vv; setivalue(&vv, v);
        luaV_finishset(L, vt, &vk, &vv, slot);
    }
}
```

Internal Lua headers

Table is saved in the Lua stack

Directly access the array part of the table

No need to call the GC in the end

Thank you!

- Follow our work in progress at https://www.github.com/titan-lang
- Email me at hgualandi@inf.puc-rio.br