Writing Csound Opcodes in Lua

Michael Gogins

Irreducible Productions
New York

Csound Conference, 2011

Outline

1. Motivation
   - Make Csound a better programming environment
   - Limitations of existing solutions
2. Lua opcodes
   - Design goals
   - Implementation
   - Results

The orchestra language

- One of Csound's main purposes is user programmability. Unfortunately, Csound is not that easy to program.
- The orchestra language is designed first for musical usefulness, second for runtime efficiency, third for easy implementation, and last of all for ease of use.
- So, it has an assembler-like syntax, incomplete lexical scoping, and no user-definable data structures except for function tables.
- The orchestra language also is full of too many clever hacks.
- So, Csound code is hard to write.

Plugin opcodes

- Plugin opcodes can be written in C or C++ — the leading general-purpose systems programming languages.
- I have contributed a header-file only C++ base class that greatly simplifies writing opcodes in C++, which I use for all my own opcodes (including these Lua Opcodes).
- So, I expected many people would contribute plugin opcodes. Why has this not happened?
- Most Csound users are not C or C++ programmers.
- Even those who know C or C++ seem reluctant to set up a build system.
- Two other solutions exist: the Python opcodes and user-defined opcodes (UDOs).

The Python opcodes

- The Python opcodes provide various means for calling into Python code from Csound instrument definitions.
- Any amount of Python code can be embedded as string literals in a Csound orchestra file.
- Python is very powerful language, but it is not efficient.
- There are quite a few Python opcodes, and that may be confusing to beginners.
- You can call any Python function, but you can't directly define an opcode in Python.
- I do not know how widely used the Python opcodes are.

User-defined opcodes

- User-defined opcodes work by plugging what is essentially the body of a Csound instrument definition into the plumbing of an opcode definition.
- Unlike plugin opcodes and the Python opcodes, UDOs are quite popular.
- UDOs are as efficient as any other Csound orchestra code, which is much faster than Python.
- The only real drawback of UDOs is... you have to write them in Csound orchestra code.
- In my opinion, that's a pretty big drawback!

Design goals

- Make Csound much easier to program and extend, using a widely known, general-purpose programming language with a simple syntax.
- Make it possible to define Csound opcodes, of any type, directly in Lua.
- As Lua is a dynamic language, no build system is required.
- Run fast.
- Run *really fast.*

What is Lua?

- Lua is Portuguese for “moon.”
- Lua is a lightweight, efficient dynamic programming language.
- Lua is specifically designed both for embedding in C/C++, and for extending with C/C++, using a stack-based calling mechanism.
- Lua provides a compact toolkit of language features such as tables, metatables, and closures, with which many styles of object-oriented and functional programming may be implemented.
- Lua has a simple yet flexible syntax, and is only slightly harder than Python to learn and to write.

LuaJIT

- Lua is already one of the fastest dynamic languages — but LuaJIT by Mike Pall goes much further.
- LuaJIT gives Lua a just-in-time optimizing trace compiler for Intel architectures.
- LuaJIT includes an efficient foreign function interface (FFI) with the ability to define C arrays, structures, and other types in Lua, and associate Lua functions with them.
- The efficiency of LuaJIT/FFI ranges from several times as fast as Lua, to faster (in certain contexts) than optimized C.
- In my opinion, LuaJIT/FFI is a glimpse of the future of programming languages! If only it had built-in parallel constructs...

How it works

- The lua_opdef opcode is an opcode that defines other opcodes.
- The user provides an opcode name and a block of Lua code.
- The Lua code declares the opcode structure in C using FFI, and defines the opcode subroutines in Lua.
- The lua_opcall set of opcodes is used to call the Lua opcodes in performance.
- Any number of output and input parameters of any type may be used, but they all come on the right hand side of the opcode name.
- Although the Lua virtual machine is single-threaded, the Lua opcodes maintain one LuaJIT state per Csound thread.

Listing 1: Orchestra Header

```
lua_opdef "luatest", {{
local ffi = require("ffi")
-- This defines the outargs/inargs part of the opcode struct.
ffi.cdef[[
    struct luatest_args_t
    {
      double *iout;
      double *arg;
    };
]]
-- This defines the iopadr.
function luatest_init(csound, opcode, carguments)
    -- Typecast carguments (an address) to a local variable (a typed pointer).
    local arguments = ffi.cast("struct luatest_args_t *", carguments)
    -- In LuaJIT FFI, dereference pointers by treating them as arrays.
    arguments.iout[0] = arguments.arg[0] * 2
    return 0
end
}}
```

Listing 2: Instrument Definition

```
instr 1
        iarg = 2
        iresult = 0
        lua_iopcall "luatest", iresult, iarg
        print iresult
endin
```

How it runs

- The proof of concept is a port of Csound's native `moogladder` filter opcode, written in C by Victor Lazzarini, to LuaJIT/FFI.
- Getting the LuaJIT/FFI code to run took some experimentation to get around idiosyncracies of the LuaJIT virtual machine!
- The LuaJIT/FFI opcode runs in 118% of the time of the native version, and sounds the same.
- The LuaJIT/FFI opcode runs in 40% of the time of a user-defined opcode version, also written by Victor Lazzarini.
- *The LuaJIT/FFI code runs almost as fast as C code, and two and a half times as fast as Csound orchestra code.*

Listing 3: Orchestra header for the Lua moogladder

```
    lua_opdef   "moogladder", {{
local ffi = require("ffi")
local math = require("math")
local string = require("string")
local csoundApi = ffi.load('csound64.dll.5.2')
ffi.cdef[[
    int csoundGetKsmps(void *);
    double csoundGetSr(void *);
    struct moogladder_t {
      double *out;
      double *inp;
      double *freq;
      double *res;
      double *istor;
      double sr;
      double ksmps;
      double thermal;
      double f;
      double fc;
      double fc2;
      double fc3;
      double fcr;
      double acr;
      double tune;
      double res4;
      double input;
      double i;
      double j;
      double k;
      double kk;
```

```
      double stg[6];
      double delay[6];
      double tanhstg[6];
    };
]]

local moogladder_ct = ffi.typeof('struct moogladder_t *')

function moogladder_init(csound, opcode, carguments)
    local p = ffi.cast(moogladder_ct, carguments)
    p.sr = csoundApi.csoundGetSr(csound)
    p.ksmps = csoundApi.csoundGetKsmps(csound)
    if p.istor[0] == 0 then
        for i = 0, 5 do
            p.delay[i] = 0.0
        end
        for i = 0, 3 do
            p.tanhstg[i] = 0.0
        end
    end
    return 0
end

function moogladder_kontrol(csound, opcode, carguments)
    local p = ffi.cast(moogladder_ct, carguments)
    -- transistor thermal voltage
    p.thermal = 1.0 / 40000.0
    if p.res[0] < 0.0 then
        p.res[0] = 0.0
    end
    -- sr is half the actual filter sampling rate
    p.fc = p.freq[0] / p.sr
```

```
p.f = p.fc / 2.0
p.fc2 = p.fc * p.fc
p.fc3 = p.fc2 * p.fc
-- frequency & amplitude correction
p.fcr = 1.873 * p.fc3 + 0.4955 * p.fc2 - 0.6490 * p.fc + 0.9988
p.acr = -3.9364 * p.fc2 + 1.8409 * p.fc + 0.9968
-- filter tuning
p.tune = (1.0 - math.exp(-(2.0 * math.pi * p.f * p.fcr))) / p.thermal
p.res4 = 4.0 * p.res[0] * p.acr
-- Nested 'for' loops crash, not sure why.
-- Local loop variables also are problematic.
-- Lower-level loop constructs don't crash.
p.i = 0
while p.i < p.ksmps do
    p.j = 0
    while p.j < 2 do
        p.k = 0
        while p.k < 4 do
            if p.k == 0 then
                p.input = p.inp[p.i] - p.res4 * p.delay[5]
                p.stg[p.k] = p.delay[p.k] + p.tune * (math.tanh(p.input * p.
                      thermal) - p.tanhstg[p.k])
            else
                p.input = p.stg[p.k - 1]
                p.tanhstg[p.k - 1] = math.tanh(p.input * p.thermal)
                if p.k < 3 then
                    p.kk = p.tanhstg[p.k]
                else
                    p.kk = math.tanh(p.delay[p.k] * p.thermal)
                end
                p.stg[p.k] = p.delay[p.k] + p.tune * (p.tanhstg[p.k - 1] - p.kk)
            end
```

```
                p.delay[p.k] = p.stg[p.k]
                p.k = p.k + 1
            end
            -- 1/2-sample delay for phase compensation
            p.delay[5] = (p.stg[3] + p.delay[4]) * 0.5
            p.delay[4] = p.stg[3]
            p.j = p.j + 1
        end
        p.out[p.i] = p.delay[5]
        p.i = p.i + 1
    end
    return 0
end
}}
```

Listing 4: Calling the Lua moogladder

```
instr 4
                prints        "Lua moogladder.\n"
    kres        init          1
    istor       init          0
    kfe         expseg        500, p3*0.9, 1800, p3*0.1, 3000
    kenv        linen         10000, 0.05, p3, 0.05
    asig        buzz          kenv, 100, sr/(200), 1
    afil        init          0
                lua_ikopcall    "moogladder", afil, asig, kfe, kres, istor
                out           afil
endin
```

Additional possibilities

- Using LuaJIT's FFI, you can use the Csound API to call back into the running host instance of Csound.
- In this way, you can do in Lua anything the Csound API will let you do.
- You can embed score-generating Lua code into a Csound orchestra.
- You can import and use any third party Lua extension or library.
- God knows what you can do!

Summary

- You can write any number of opcodes in a real programming language.
- In your opcodes you can define additional functions, tables, structures, and classes.
- You can use lambdas and closures.
- In your opcodes you can call any C function in a shared library. That includes all system calls. That includes Csound itself.
- Your opcodes will run *very fast*.
- On Windows for x86, LuaJIT already comes with Csound; on Linux for x86, LuaJIT is download, make, and install.

Outlook

- It might be possible to define a completely new software synthesizer written for LuaJIT that would, via FFI, use all Csound opcodes — including all existing opcodes, and any new opcodes that are plugins.

For further reading I

Lua.org.
The Programming Language Lua.
http://www.lua.org
Pontifícia Universidade Católica do Rio de Janeiro.

R. Ierusalimschy, L. H. de Figueiredo, W. Celes.
Lua 5.1 Reference Manual.
Lua.org, 2006

R. Ierusalimschy.
Programming in Lua.
Lua.org, 2003.

For further reading II

Mike Pall.
The LuaJIT Project.
http://luajit.org.