Writing Csound Opcodes in Lua

Michael Gogins
michael.gogins@gmail.com

Lua is a fast, lightweight, dynamic language designed for embedding in applications. With Mike Pall's LuaJIT, Lua gains a just-in-time optimizing trace compiler with almost the execution speed of C. For some time, Csound has included a Lua interface to the Csound API. Now, the `lua_opdef` and `lua_opcall` opcodes enable writing Csound opcodes directly in Lua. The Lua code is embedded in the Csound orchestra, and compiled and optimized at run time. Users can write Csound code in a modern programming language with simple syntax; full lexical scoping; support for classes, lambdas and closures; and access to a variety of third party libraries, including *all* shared libraries with exported C interfaces. It's even possible to call back into the Csound API from Lua. A performance analysis compares Csound's Moog ladder filter written in C, as a user-defined opcode, and as a Lua opcode. The Lua opcode runs in 118% of the time of native C and in 40% of the time of the user-defined opcode.

Introduction

Csound (Vercoe *et al*., 2011) is a user-programmable software synthesizer with a powerful library of signal processing routines, or "opcodes." Csound itself is mostly written in C, but users program Csound in the "orchestra language" to define instruments, and in the "score language," to define notes to be rendered by the instruments. We are concerned here only with the orchestra language.

The musical semantics of Csound are excellent. It computes audio at any sampling rate, with floating point samples, for any number of channels. Csound renders dense score events, automatically creates instances of instruments for polyphony, and handles variant tunings, tied notes, tempo changes, *etc*.

However, the orchestra language seems created first for musical usefulness; second, for runtime speed; third, for ease of implementing Csound itself; and only last, for ease of writing code. As a result, the orchestra language has an assembler-like syntax that mixes operations on control-rate variables and audio-rate variables. It lacks lexical scoping, classes, or user-defined structures except function tables. To work around these limitations, many hacks have been used to extend the orchestra language.

As a result, Csound code is not so easy to write. Several solutions to this problem have been implemented: plugin opcodes, the suite of Python opcodes, and user-defined opcodes (UDOs). As discussed below, none of these solutions are completely successful.

The Lua opcodes presented here are designed to overcome the limitations of these alternatives.

I. Alternatives to Csound Code

Plugin Opcodes

Csound was designed from the start to be extensible with new opcodes written in C. In the past, users who wrote their own opcodes had to recompile Csound itself. The newer plugin facility enables opcodes to be compiled independently. Csound automatically discovers, loads, and runs such plugins during performance. They can be written in C or C++, which remain the leading systems programming languages. I've contributed a header-file only template class library to simplify writing opcodes in C++.

So, I expected many people would write plugin opcodes. Why haven't they? Csound users are not necessarily experienced C or C++ programmers. Perhaps more importantly, it can take a long time to debug C or C++ code through the necessary cycles of write/build/run/fail/rewrite.

The Python Opcodes

The Python opcodes enable calling into Python code from Csound orchestras. Any amount of Python code can be embedded as string literals in Csound code.

You can call any Python function, but you can't directly define an opcode in Python. There are quite a few Python opcodes, and that may be confusing. Python is a very powerful language, but Python code typically executes 3 to 100 times slower than C++, on average 30 times slower. That's fine for score generation, but slow for signal processing. I don't know how widely used the Python opcodes are.

User-Defined Opcodes

User-defined opcodes work by plugging a body like a Csound instrument definition into the plumbing of an opcode definition. Unlike plugin opcodes and the Python opcodes, UDOs are popular. They are as efficient as any other Csound orchestra code, *i.e.* much faster than Python.

The only real drawback of UDOs is... you have to write them in Csound orchestra code! So, while UDOs are great for developing reusable libraries of Csound code, they are not really a solution to the difficulty of writing Csound code.

II. Using Lua in Csound

The design goals of the Lua opcodes are:

1. Make it possible to write Csound code in a user-friendly, high-level language with full lexical scoping, structures and classes, and support for functional programming:
    1. Execute arbitrary blocks of Lua code at i-rate (`lua_exec`). Such code can define global functions, classes, and so on to be used by Lua opcodes, or do anything else for that matter.
    2. Directly define opcodes in Lua for any processing rate, any types of parameters, and any number of output and input parameters (`lua_opdef`).
    3. Effciently call these new opcodes defined in Lua (`lua_iopcall`, `lua_ikopcall`, `lua_iaopcall`).
2. Require the installation of no third party software packages, or at least a minimum installation.
3. Require no build system or external compilation.
4. Run *really fast*.

What is Lua?

Lua is Portuguese for "moon." And Lua (Lua.org, 2011; Ierusalimschy *et al*. 2003; Ierusalimschy 2006) is a lightweight, efficient dynamic programming language, designed for embedding in C/C++ and extending with C/C++. Lua has a stack-based calling mechanism and provides a toolkit of features (tables, metatables, anonymous functions, and closures) with which many styles of object-oriented and functional programming may be implemented. Lua's syntax is only slightly harder than Python's.

Lua is already one of the fastest dynamic languages — yet LuaJIT by Mike Pall (2011) goes much

further, giving 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. The speed of LuaJIT/FFI ranges from several times as fast as Lua, to faster (in some contexts) than optimized C. In my opinion, LuaJIT/FFI is a glimpse of the future of programming languages!

How It Works

Here we describe only the `lua_opdef` and `lua_ikopcall` opcodes; the others work the same way.

The `lua_opdef` opcode (Listing 1) is used to create Lua opcodes in the Csound orchestra header. The user provides an opcode name and a block of Lua code in a multi-line string literal enclosed by double braces (Listing 3). `lua_opdef` has only an init function (Listing 1, lines 7-57), which first retrieves the name of the Lua opcode, then executes the user-defined Lua code (in the `luaL_dostring` call, line 20). This Lua code must first use a LuaJIT/FFI `cdef` to declare the Lua opcode's structure in C (Listing 3, lines 6-33), then define the actual Lua opcode subroutines (lines 37-102). `lua_opdef`'s init routine then retrieves and stores Lua references to the newly defined subroutines (Listing 1, lines 22-52), as it is faster to call Lua functions from C by reference than by name.

The Lua virtual machine is single-threaded, so the Lua opcodes maintain a separate LuaJIT state for each Csound thread. The `manageLuaState` and `manageLuaReferenceKeys` functions (Listing 1, lines 15 and 16) are thread-safe global functions for efficiently creating and accessing LuaJIT virtual machines and sets of references to Lua functions. They exist so that as Csound becomes more and more commonly used on multi-core computers, the Lua opcodes also will support parallelization for increased performance.

There are four possible opcode subroutine signatures, where `opname` is the associated opcode name:

```
function opname_init(csound, opcode, carguments) … end
function opname_kontrol(csound, opcode, carguments) … end
function opname_audio(csound, opcode, carguments) … end
function opname_noteoff(csound, opcode, carguments)… end
```

`csound` is a Lua lightuserdata, *i.e.* a C pointer, which points to the running instance of Csound. `opcode` is a lightuserdata that points to the `lua_opcall` opcode structure. `carguments` is a lightuserdata that points to the start of the `arguments` section of the `lua_opcall` opcode structure. `arguments` is a large array of pointers to `MYFLT` (all Csound C opcode arguments are `MYFLT *`).

The `lua_opcall` set of opcodes (Listing 2) is used to call the Lua opcodes in performance. Any number of output and input parameters of any type may be used, all on the right hand side of the opcode name (see Listing 4, line 8). All output values are passed back to Csound in the arguments.

`lua_iopcall` is called only at `i`-rate, and requires only an `opname_init` function. `lua_ikopcall` is called at `i`-rate and at `k`-rate, and requires an `opname_init` function and an `opname_kontrol` function. `lua_iaopcall` is called at `i`-rate and at `a`-rate requires `opname_init` and `opname_audio`. `lua_iopcall_off`, `lua_ikopcall_off`, and `lua_iaopcall_off` are identical to `lua_iopcall`, `lua_ikopcall`, and `lua_iaopcall`, respectively, except that in addition they require an `opname_noteoff` function that Csound calls when the instrument running the opcode is inactivated.

All each opcode subroutine does is retrieve the LuaJIT virtual machine that belongs to the calling thread, retrieve the Lua reference to the Lua opcode subroutine, push the arguments on the Lua virtual machine's stack, execute the protected call that has been set up on the stack, and pop the return value off the stack (e.g. lines 33-41 in Listing 2).

The critical concept in the Lua opcodes is the opcode structure. It is what enables Csound to pass opcode arguments to Lua code and get return values back. When Csound calls a `lua_opcall` opcode, the actual memory is created by Csound based on the size of the `lua_opcall` opcode structure, which contains a large `arguments` array. Because the outtypes to `lua_opcall` are declared as Csound type `N`, *i.e.* any number of arguments of any type, Csound simply copies the address of each argument into the corresponding slot of the `arguments` array, without type checking. As far as LuaJIT is concerned, the opcode structure is an FFI `cdef` that defines a C structure. When the Lua opcode functions are called, the last parameter contains the address of this arguments array. LuaJIT must FFI to type cast this address as a pointer to the Lua opcode structure (Listing 3, lines 35, 38, and 51). In effect, the arguments array and the Lua opcode structure form a union – two different types that refer to the same block of memory. As long as the types of the arguments passed agree with the types declared in the opcode structure, the arguments and any other members can be efficiently accessed as Lua types.

Listing 3, lines 39-40 shows a Lua opcode using the Csound API via LuaJIT's FFI to call back into Csound to obtain and store the sampling rate and number of sample frames per kperiod.

III. Results

The proof of concept is a port of Csound's `moogladder` filter opcode, written in C by Victor Lazzarini, to LuaJIT/FFI. The Csound orchestra header code for the Lua opcode is shown in Listing 3, and the Csound instrument definition that uses the opcode is shown in Listing 4.

Results of using `moogladder` with a moving center frequency to filter a buzz instrument are shown in Table 1. All three implementations of `moogladder` sound the same. Only the time spent in the actual `moogladder` opcode is counted. This was run on a quad core Intel Core i7 720 1.6 GHz machine running Csound 5.13 double precision samples on Windows 7 Home Premium Edition.

Table 1. Comparison of Opcode Performance

Implementation	*Run 1*	*Run 2*	*Mean*	*Opcode only*	*% of "C"*	*% of UDO*	*% of Lua*
Instrument without `moogladder`	0.065	0.072	0.069	0.000	n/a	n/a	n/a
Instrument with native C `moogladder`	1.065	1.082	1.074	1.005	100.0%	33.4%	84.5%
Instrument with UDO `moogladder`	3.104	3.052	3.078	3.010	299.5%	100.0%	252.9%
Instrument with Lua `moogladder`	1.286	1.231	1.259	1.190	118.4%	39.5%	100.0%

The LuaJIT/FFI code runs almost as fast as the C code, and two and a half times as fast as Csound orchestra code.

Things to Watch Out For

Getting the `moogladder` Lua code to run took experimentation to work around idiosyncracies of LuaJIT's virtual machine. In particular, using Lua's normal `for` loop construct proved impossible – every run simply crashed. Fortunately, using the `while` construct to create `for` loops is easy, and works fine. There may be other problems with LuaJIT/FFI that different Lua code would expose.

However, I have not yet run into any problems with LuaJIT that could not be worked around.

Listings

Listing 1. `lua_opdef` *Definition.*

```
class cslua_opdef : public OpcodeBase<cslua_opdef>
{
public:
    MYFLT *opcodename_;
    MYFLT *luacode_;
public:
    int init(CSOUND *csound)
    {
        int result = OK;
        lua_State *L = manageLuaState();
        const char *opcodename =  csound->strarg2name(csound,
                                  (char *) 0, opcodename_,
                                  (char *)"default",
                                  (int) csound->GetInputArgSMask(this));
        const char *luacode =  csound->strarg2name(csound,
                               (char *) 0, luacode_,
                               (char *)"default",
                               (int) csound->GetInputArgSMask(this));
        log(csound, "Executing Lua code:\n%s\n", luacode);
        result = luaL_dostring(L, luacode);
        if (result == 0) {
            keys_t &keys = manageLuaReferenceKeys(L, opcodename);
            log(csound, "Opcode: %s\n", opcodename);
            log(csound, "Result: %d\n", result);
            char init_function[0x100];
            std::snprintf(init_function, 0x100, "%s_init", opcodename);
            lua_getglobal(L, init_function);
            if (!lua_isnil(L, 1)) {
                keys.init_key = luaL_ref(L, LUA_REGISTRYINDEX);
                lua_pop(L, 1);
            }
            char kontrol_function[0x100];
            std::snprintf(kontrol_function, 0x100, "%s_kontrol", opcodename);
            lua_getglobal(L, kontrol_function);
            if (!lua_isnil(L, 1)) {
                keys.kontrol_key = luaL_ref(L, LUA_REGISTRYINDEX);
                lua_pop(L, 1);
            }
            char audio_function[0x100];
            std::snprintf(audio_function, 0x100, "%s_audio", opcodename);
            lua_getglobal(L, audio_function);
            if (!lua_isnil(L, 1)) {
                keys.audio_key = luaL_ref(L, LUA_REGISTRYINDEX);
                lua_pop(L, 1);
            }
            char noteoff_function[0x100];
            std::snprintf(noteoff_function, 0x100, "%s_noteoff", opcodename);
            lua_getglobal(L, noteoff_function);
            if (!lua_isnil(L, 1)) {
                keys.noteoff_key = luaL_ref(L, LUA_REGISTRYINDEX);
                lua_pop(L, 1);
            }
        } else {
            log(csound, "Failed with: %d\n", result);
        }
        return result;
    }
};
```

Listing 2. `lua_opcall` *Definition.*

```
class cslua_opcall: public OpcodeBase<cslua_opcall>
{
public:
    MYFLT *opcodename_;
    MYFLT *arguments[0x3000];
    const char *opcodename;
public:
    int init(CSOUND *csound)
```

```
9       {
10          int result = OK;
11          opcodename = csound->strarg2name(csound, (char *) 0,
12                                           opcodename_,
13                                           (char *)"default",
14                                           (int) csound->GetInputArgSMask(this));
15          lua_State *L = manageLuaState();
16          keys_t &keys = manageLuaReferenceKeys(L, opcodename);
17          lua_rawgeti(L, LUA_REGISTRYINDEX, keys.init_key);
18          lua_pushlightuserdata(L, csound);
19          lua_pushlightuserdata(L, this);
20          lua_pushlightuserdata(L, &arguments);
21          if (lua_pcall(L, 3, 1, 0) != 0) {
22              log(csound, "Lua error in \"%s_init\": %s.\n", opcodename, lua_tostring(L, -1));
23          }
24          result = lua_tonumber(L, -1);
25          lua_pop(L, 1);
26          return OK;
27      }
28      int kontrol(CSOUND *csound)
29      {
30          int result = OK;
31          lua_State *L = manageLuaState();
32          keys_t &keys = manageLuaReferenceKeys(L, opcodename);
33          lua_rawgeti(L, LUA_REGISTRYINDEX, keys.kontrol_key);
34          lua_pushlightuserdata(L, csound);
35          lua_pushlightuserdata(L, this);
36          lua_pushlightuserdata(L, &arguments);
37          if (lua_pcall(L, 3, 1, 0) != 0) {
38              log(csound, "Lua error in \"%s_kontrol\": %s.\n", opcodename, lua_tostring(L, -1));
39          }
40          result = lua_tonumber(L, -1);
41          lua_pop(L, 1);
42          return result;
43      }
44      int audio(CSOUND *csound)
45      {
46          int result = OK;
47          lua_State *L = manageLuaState();
48          keys_t &keys = manageLuaReferenceKeys(L, opcodename);
49          lua_rawgeti(L, LUA_REGISTRYINDEX, keys.audio_key);
50          lua_pushlightuserdata(L, csound);
51          lua_pushlightuserdata(L, this);
52          lua_pushlightuserdata(L, arguments);
53          if (lua_pcall(L, 3, 1, 0) != 0) {
54              log(csound, "Lua error in \"%s_audio\": %s.\n", opcodename, lua_tostring(L, -1));
55          }
56          result = lua_tonumber(L, -1);
57          lua_pop(L, 1);
58          return result;
59      }
60  };
```

Listing 3. `moogladder` *Opcode Definition.*

```
1   lua_opdef "moogladder", {{
2   local ffi = require("ffi")
3   local math = require("math")
4   local string = require("string")
5   local csoundApi = ffi.load('csound64.dll.5.2')
6   ffi.cdef[[
7       int csoundGetKsmps(void *);
8       double csoundGetSr(void *);
9       struct moogladder_t {
10        double *out;
11        double *inp;
12        double *freq;
13        double *res;
14        double *istor;
15        double sr;
16        double ksmps;
17        double thermal;
18        double f;
19        double fc;
20        double fc2;
```

```
21       double fc3;
22       double fcr;
23       double acr;
24       double tune;
25       double res4;
26       double input;
27       double i, j, k;
28       double kk;
29       double stg[6];
30       double delay[6];
31       double tanhstg[6];
32     };
33 ]]
34
35 local moogladder_ct = ffi.typeof('struct moogladder_t *')
36
37 function moogladder_init(csound, opcode, carguments)
38     local p = ffi.cast(moogladder_ct, carguments)
39     p.sr = csoundApi.csoundGetSr(csound)
40     p.ksmps = csoundApi.csoundGetKsmps(csound)
41     if p.istor[0] == 0 then
42         for i = 0, 5 do
43             p.delay[i] = 0.0
44             p.tanhstg[i] = 0.0
45         end
46     end
47     return 0
48 end
49
50 function moogladder_kontrol(csound, opcode, carguments)
51     local p = ffi.cast(moogladder_ct, carguments)
52     -- transistor thermal voltage
53     p.thermal = 1.0 / 40000.0
54     if p.res[0] < 0.0 then
55         p.res[0] = 0.0
56     end
57     -- sr is half the actual filter sampling rate
58     p.fc = p.freq[0] / p.sr
59     p.f = p.fc / 2.0
60     p.fc2 = p.fc * p.fc
61     p.fc3 = p.fc2 * p.fc
62     -- frequency & amplitude correction
63     p.fcr = 1.873 * p.fc3 + 0.4955 * p.fc2 - 0.6490 * p.fc + 0.9988
64     p.acr = -3.9364 * p.fc2 + 1.8409 * p.fc + 0.9968
65     -- filter tuning
66     p.tune = (1.0 - math.exp(-(2.0 * math.pi * p.f * p.fcr))) / p.thermal
67     p.res4 = 4.0 * p.res[0] * p.acr
68     -- Nested 'for' loops crash, not sure why.
69     -- Local loop variables also are problematic.
70     -- Lower-level loop constructs don't crash.
71     p.i = 0
72     while p.i < p.ksmps do
73         p.j = 0
74         while p.j < 2 do
75             p.k = 0
76             while p.k < 4 do
77                 if p.k == 0 then
78                     p.input = p.inp[p.i] - p.res4 * p.delay[5]
79                     p.stg[p.k] = p.delay[p.k] + p.tune * (math.tanh(p.input * p.thermal) -
   p.tanhstg[p.k])
80                 else
81                     p.input = p.stg[p.k - 1]
82                     p.tanhstg[p.k - 1] = math.tanh(p.input * p.thermal)
83                     if p.k < 3 then
84                         p.kk = p.tanhstg[p.k]
85                     else
86                         p.kk = math.tanh(p.delay[p.k] * p.thermal)
87                     end
88                     p.stg[p.k] = p.delay[p.k] + p.tune * (p.tanhstg[p.k - 1] - p.kk)
89                 end
90                 p.delay[p.k] = p.stg[p.k]
91                 p.k = p.k + 1
92             end
93             -- 1/2-sample delay for phase compensation
```

```
94              p.delay[5] = (p.stg[3] + p.delay[4]) * 0.5
95              p.delay[4] = p.stg[3]
96              p.j = p.j + 1
97          end
98          p.out[p.i] = p.delay[5]
99          p.i = p.i + 1
100        end
101        return 0
102     end
103     }}
```

Listing 4. `moogladder` *Invocation.*

```
1  instr 4
2       kres        init        1
3       istor       init        0
4       kfe         expseg      500, p3*0.9, 1800, p3*0.1, 3000
5       kenv        linen       10000, 0.05, p3, 0.05
6       asig        buzz        kenv, 100, sr/(200), 1
7       afil        init        0
8                   lua_ikopcall    "moogladder", afil, asig, kfe, kres, istor
9                   out         afil
10 endin
```

References

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

Ierusalimschy, R. March 2006. ***Programming in Lua***, 2nd edition. Lua.org.

Lua.org, Pontifícia Universidade Católica do Rio de Janeiro. Retrieved August 18, 2011. ***The Programming Language Lua***. <http://www.lua.org>.

Pall, Mike. Retrieved August 18, 2011. ***The LuaJIT Project***. <http://luajit.org>

Vercoe, Barry, et al. Retrieved August 18, 2011. ***The Canonical Csound Reference Manual***, Version 5.13. <http:csounds.com>.