Foreign Function Interface & Lua

Foreign Function Interface & Lua

Calling code from shared libraries in C is simple:
the canonical way is look at the interface

(`somelib.h`)

```
/* somelib.h */
#ifndef SOMELIB_H
#define SOMELIB_H

typedef struct {
    int num;
    double dnum;
} DataPoint;

DataPoint *add_data(const DataPoint *dps, unsigned n);

#endif /* SOMELIB_H */
```

Foreign Function Interface & Lua

(somelib.c)

```
/* somelib.c */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "somelib.h"

DataPoint *add_data(const DataPoint* dps, unsigned n) {
    DataPoint *out;
    out = malloc(sizeof(DataPoint));
    assert(out); /* hmm, we are drastic here... */
    out->num = 0;
    out->dnum = 0.0;
    for (unsigned i = 0; i < n; ++i) {
        out->num += dps[i].num;
        out->dnum += dps[i].dnum;
    }
    return out;
}
```

Foreign Function Interface & Lua

using the interface :

```
#include <stdio.h>
#include <stdlib.h>
#include "somelib.h"

int main(void)
{
    DataPoint *dout;
    DataPoint dp[4] = {{2, 2.2}, {3, 3.3}, {4, 4.4}, {5, 5.5}};

    printf("Calling add_data\n");
    dout = add_data(dp, sizeof(dp) / sizeof(DataPoint));

    if (dout){
       printf("dout = {%d, %lf}\n", dout->num, dout->dnum);
        free(dout);
    }
    return 0;
}
```

Foreign Function Interface & Lua

link the shared library at building time:

```
$> gcc -fPIC -shared -I. -Wall -Wextra -Wunused -Wimplicit -Wreturn-type
  -Wdeclaration-after-statement -Wno-unknown-pragmas -Wmissing-prototypes
  -Wmissing-declarations -Wparentheses -Wswitch -Wtrigraphs -Wpointer-arith
  -Wcast-qual -Wcast-align -Wwrite-strings -Wold-style-definition
  -o somelib.so somelib.c

$> gcc -I. -L. -Wall -Wextra -Wunused -Wimplicit -Wreturn-type
  -Wdeclaration-after-statement -Wno-unknown-pragmas -Wmissing-prototypes
  -Wmissing-declarations -Wparentheses -Wswitch -Wtrigraphs
  -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings
  -Wold-style-definition
  -Wl,-rpath,'$ORIGIN/.' test-043.c -o test-043 -l:somelib.so

$> ./test-043
Calling add_data
dout = {14, 15.400000}
```

Foreign Function Interface & Lua

Using `dlopen()` to load shared library *at run-time* (1/3):

```
#include <dlfcn.h>
#include <ffi.h>
#include <stdio.h>
#include <stdlib.h>
#include "somelib.h"

int main(void) {
    void *libhandle; void *add_data_fn; char *err;
    ffi_type *args[2];ffi_type *rtype; ffi_cif cif;
    ffi_status status;
    DataPoint dp[4] = {{2, 2.2}, {3, 3.3}, {4, 4.4}, {5, 5.5}};
    DataPoint *pdp;
    unsigned nelems; void *values[2];
    DataPoint *dout;

    libhandle = dlopen("./somelib.so", RTLD_LAZY); /* <--- string ! */
    if (!libhandle) {
        fprintf(stderr, "dlopen error: %s\n", dlerror());
        exit(1);
    }
    add_data_fn = dlsym(libhandle, "add_data"); /* <--- string ! */
    err = dlerror();
    if (err) {
        fprintf(stderr, "dlsym failed: %s\n", err);
        exit(1);
    }
:
```

Foreign Function Interface & Lua

Using `dlopen()` to load shared library *at run-time* (2/3):

```
    :

    args[0]=&ffi_type_pointer; args[1]=&ffi_type_uint;
    rtype = &ffi_type_pointer;

    status = ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 2, rtype, args);
    if (status != FFI_OK) {
        fprintf(stderr, "ffi_prep_cif failed: %d\n", status);
        exit(1);
    }

    pdp = dp;
    nelems = sizeof(dp) / sizeof(DataPoint);
    values[0] =  &pdp;
    values[1] =  &nelems;
    printf("Calling add_data via libffi\n");
    dout = NULL;
    ffi_call(&cif, FFI_FN(add_data_fn), &dout, values);
    if (dout) {
       printf("dout = {%d, %lf}\n", dout->num, dout->dnum);
    }
    return 0;
}
```

Foreign Function Interface & Lua

Using `dlopen()` to load shared library *at run-time* (3/3):

```
$>gcc test-044a.c -o test-044a -lffi -ldl
```

vs.

```
$> gcc -I. -L.  -Wl,-rpath,'$ORIGIN/.' test-043.c
  -o test-043 -l:somelib.so
```

Apparently no gain: `libffi` vs `somelib` shared library.
Not only: code is more complex.
But the key point is *run-time*: both library and function are specified by strings.

Foreign Function Interface & Lua

run-time

⤷ *scripting language*

⤷ *Lua*

⤷ LuaTEX

It is (seems) possible to use a shared library *without* a Lua binding (i.e. another shared library) as done with SWIGLIB.

Foreign Function Interface & Lua

Python already supports `libffi` in two ways :

1) Python: ctypes

```
from ctypes import cdll, Structure, c_int, c_double, c_uint

lib = cdll.LoadLibrary('./somelib.so')
print('Loaded lib {0}'.format(lib))

class DataPoint(Structure):
    _fields_ = [('num', c_int),
                ('dnum', c_double)]

dps = (DataPoint * 4)((2, 2.2), (3, 3.3), (4, 4.4), (5, 5.5))

add_data_fn = lib.add_data
add_data_fn.restype = DataPoint

print('Calling add_data via ctypes')
dout = add_data_fn(dps, 4)
print('dout = {0}, {1}'.format(dout.num, dout.dnum))
```

ctypes looks similar to the previous `libffi` (complex) code.

Foreign Function Interface & Lua

2) Python: cffi

```
from cffi import FFI

ffi = FFI()

lib = ffi.dlopen('./somelib.so')
print('Loaded lib {0}'.format(lib))

# ---> Describe the data type and function prototype to cffi. <----
ffi.cdef('''
typedef struct {
    int num;
    double dnum;
} DataPoint;

DataPoint add_data(const DataPoint* dps, unsigned n);
''')

dps = ffi.new('DataPoint[]', [(2, 2.2), (3, 3.3), (4, 4.4), (5, 5.5)])

print('Calling add_data via cffi')

dout = lib.add_data(dps, 4)
print('dout = {0}, {1}'.format(dout.num, dout.dnum))
```

it looks similar to the original C code !

Foreign Function Interface & Lua

So...why don’t we use the ‘cffi way’ *always* ?

And why don’t we use `libffi` *always* ?

Foreign Function Interface & Lua

First:

parsing C declarations requires a full C parser that must be integrated with the interpreter of the scripting language... not an easy task.

Foreign Function Interface & Lua

Second:

> “In terms of its implementation, libffi does as much as possible in portable C, but eventually has to resort to **assembly routines** written for each architecture and calling convention it supports. There routines perform the actual register and stack modifications around the call to the given function to make sure the call conforms to the calling convention. Note also that due to this extra work, calls via libffi are much **slower** than direct calls created by the compiler.”

`(Ref. https://eli.thegreenplace.net/2013/03/04/flexible-runtime-interface-to-shared-libraries-with-libffi)`

Foreign Function Interface & Lua

`libffi` has **assembly routines** for

(AArch64 (ARM64), iOS), (AArch64, Linux), (Alpha, Linux), (Alpha, Tru64), (ARC, Linux), (ARM, Linux), (ARM, iOS), (AVR32, Linux), (Blackfin, uClinux), (HPPA, HPUX), (IA-64, Linux), (M68K, FreeMiNT), (M68K, Linux), (M68K, RTEMS), (M68K, OpenBSD/mvme88k), (Meta, Linux), (MicroBlaze, Linux), (MIPS, IRIX), (MIPS, Linux), (MIPS RTEMS), (MIPS64, Linux), (Moxie, Bare-metal), (Nios II, Linux), (OpenRISC, Linux), (PowerPC 32-bit, AIX), (PowerPC 64-bit, AIX), (PowerPC AMIGA), (PowerPC 32-bit, Linux), (PowerPC 64-bit, Linux), (PowerPC Mac, OSX), (PowerPC 32-bit, FreeBSD), (PowerPC 64-bit, FreeBSD), (S390, Linux), (S390X, Linux), (SPARC, Linux), (SPARC, Solaris), (SPARC64, Linux), (SPARC64, FreeBSD), (SPARC64, Solaris), (TILE-Gx/TILEPro, Linux), (VAX, OpenBSD/vax), (X86, FreeBSD), (X86, GNU, HURD), (X86, Interix), (X86, kFreeBSD), (X86, Linux), (X86, Mac, OSX), (X86, OpenBSD), (X86, OS/2), (X86, Solaris), (X86, Windows/Cygwin), (X86, Windows/MingW), (X86-64, FreeBSD), (X86-64 Linux), (X86-64, Linux/x32), (X86-64 OpenBSD), (X86-64, Solaris), (X86-64 Windows/Cygwin), (X86-64, Windows/MingW), (Xtensa, Linux).

Foreign Function Interface & Lua

TEXLive current platforms:

aarch64-linux, amd64-freebsd, amd64-netbsd, armhf-linux, i386-cygwin, i386-freebsd, i386-linux, i386-netbsd, i386-solaris, sparc-solaris, win32, x86_64-cygwin, x86_64-darwin, x86_64-darwinlegacy, x86_64-linux, x86_64-linuxmusl, x86_64-solaris.

platforms (very likely) are not a problem

Foreign Function Interface & Lua

And what about slowdown ?

> “In theory, it’s possible to use JIT-ing to dynamically generate efficient calling code once the function signature is known, but AFAIK libffi does not implement this.”

`(Ref. https://eli.thegreenplace.net/2013/03/04/flexible-runtime-interface-to-shared-libraries-with-libffi)`

It’s true that `libffi` has not JIT support but ...

Foreign Function Interface & Lua

what does JIT mean ?

1: from a single source (shared among different platforms) create machine code (specific for each platform) at run-time. It’s highly desirable that the machine code is optimized.

2: execute that machine code, also at run-time. It’s highly desirable that the execution reuses machine code as much as possible.

3: the steps above must result in a overall faster execution of the whole program, eventually disabling the JIT

Foreign Function Interface & Lua

Doesn’t the compiler do the same ? From source code as

```
/* somelib.c */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "somelib.h"

DataPoint *add_data(const DataPoint* dps, unsigned n) {
    DataPoint *out;
    out = malloc(sizeof(DataPoint));
    assert(out); /* hmm, we are drastic here... */
    out->num = 0;
    out->dnum = 0.0;
    for (unsigned i = 0; i < n; ++i) {
        out->num += dps[i].num;
        out->dnum += dps[i].dnum;
    }
    return out;
}
```

Foreign Function Interface & Lua

it produces the (assembly and then the assembler makes the) machine code:

```
$> objdump -dR  -mi386:x86-64 somelib.so
 :
00000000000006f0 <add_data>:
 6f0:  55                                push   %rbp
 6f1:  48 89 e5                          mov    %rsp,%rbp
 6f4:  48 83 ec 20                       sub    $0x20,%rsp
 6f8:  48 89 7d e8                       mov    %rdi,-0x18(%rbp)
 6fc:  89 75 e4                          mov    %esi,-0x1c(%rbp)
 6ff:  bf 10 00 00 00                    mov    $0x10,%edi
 :
 7a7:  3b 45 e4                          cmp    -0x1c(%rbp),%eax
 7aa:  72 a7                             jb     753 <add_data+0x63>
 7ac:  48 8b 45 f8                       mov    -0x8(%rbp),%rax
 7b0:  c9                                leaveq
 7b1:  c3                                retq
```

so...where is the difference ?

Foreign Function Interface & Lua

The huge difference is that a JIT compiler builds at run-time the stream of bytes and then execute them. Kind of (pseudo Lua)

```
local add_data = function(dps,n)
  :
end
-- this function magically translates  add_data into Intel machine code
local machine_code = compile (add_data)
-- machine_code = {
--    0x55,                   -- push   %rbp
--    0x48, 0x89, 0xe5,       -- mov    %rsp,%rbp
--    0x48, 0x83, 0xec, 0x20, -- sub    $0x20,%rsp
--    :
--  }

-- this function magically executes the machine code, i.e call  add_data
local my_dps={...}
local my_n = ...
local ret_val = execute(machine_code,my_dps,my_n)
```

why magically ?

Foreign Function Interface & Lua

why magically ?

1) at run-time, it's very hard to a `machine_code` stream better than original function: the translation phase takes time and the machine code produced could be slower than the original;

2) a stream of bytes created at run-time is a region of memory marked as «data», and for security reasons processors should not execute data regions. Sysadmins can enforce this rule.

Foreign Function Interface & Lua

LuaJIT(TEX):

1) has a JIT compiler for the Lua language

2) has a Foreign Function Interface

3) uses JIT for Foreign Function Interface

It’s like Python cffi but *it doesn’t use libffi !*

Foreign Function Interface & Lua

LuaJIT(TEX):

```
local ffi = require("ffi")
ffi.cdef[[
typedef struct {
    int num;
    double dnum;
} DataPoint;
DataPoint *add_data(const DataPoint *dps, unsigned n);
]]
local somelib= ffi.load( "./somelib.so")
local dp = ffi.new("DataPoint[4]")
local res = ffi.new("DataPoint[1]")
dp[0].num=2; dp[0].dnum=2.2;
dp[1].num=3; dp[1].dnum=3.3;
dp[2].num=4; dp[2].dnum=4.4;
dp[3].num=5; dp[3].dnum=5.5;
res = somelib.add_data(dp,4)
print(string.format("res.num=%s, res.dnum=%f\n",res[0].num,res[0].dnum))
```

Foreign Function Interface & Lua

LuaJIT:

```
$> luajit-2.1.0-beta3 test-somelib.lua
  res.num=14, res.dnum=15.400000
```

A core component of LuaJIT is the dynamic assembler DynASM:

1) DynASM is a pre-processing assembler: it converts mixed C/Assembler source to plain C code. The primary knowledge about instruction names, operand modes, registers, opcodes and how to encode them is only needed in the pre-processor.

2) The generated C code is extremely small and fast. A tiny embeddable C library helps with the process of dynamically assembling, relocating and linking machine code. There are no outside dependencies on other tools (such as stand-alone assemblers or linkers).

Foreign Function Interface & Lua

Lua:

The `luaffi` (https://github.com/facebookarchive/luaffifb) is the only project designed to be interface compatible with the FFI library in LuaJIT.

Pros: It uses DynASM, and the C parser from LuaJIT. It seems to work under Intel (Linux & Win 64)

Cons: not maintained anymore, old versions of DynASM and parser, it doesn't work under ARM (and perhaps OSX)

⟹ *next step is to update luaffi to current LuaJIT FFI (LuaTEX 1.09.0)*

Foreign Function Interface & Lua

LuajitTEX & LuaTEX:

```
$> luajittex  --luaonly test-somelib.lua
res.num=14, res.dnum=15.400000

$>$ luatex --luaonly test-somelib.lua
res.num=14, res.dnum=15.400000
```

It works ok, but we cannot see JIT in action here...

Foreign Function Interface & Lua

Let’s see what happens with 1 millions of calls:

```
local ffi = require("ffi")
ffi.cdef[[
typedef struct {
    int num;
    double dnum;
} DataPoint;
DataPoint *add_data(const DataPoint *dps, unsigned n);
]]
local somelib= ffi.load( "./somelib.so")
local dp = ffi.new("DataPoint[4]")
local res = ffi.new("DataPoint[1]")
-- print(dp) -- = {{2, 2.2}, {3, 3.3}, {4, 4.4}, {5, 5.5}};
dp[0].num=2; dp[0].dnum=2.2;
dp[1].num=3; dp[1].dnum=3.3;
dp[2].num=4; dp[2].dnum=4.4;
dp[3].num=5; dp[3].dnum=5.5;
for i=1,1000*1000 do
 res = somelib.add_data(dp,4)
end
print(string.format("res.num=%s, res.dnum=%f\n",res[0].num,res[0].dnum))
```

Foreign Function Interface & Lua

Let’s see what happens with 1 millions of calls:

```
$> time luatex --luaonly test-somelib.lua
res.num=14, res.dnum=15.400000
real   0m0.937s
user   0m0.918s
sys    0m0.016s

$> time luajittex --luaonly test-somelib.lua # by default no JIT !
res.num=14, res.dnum=15.400000
real   0m0.441s
user   0m0.420s
sys    0m0.020s

$> time luajittex --jiton --luaonly test-somelib.lua
res.num=14, res.dnum=15.400000
real   0m0.131s
user   0m0.130s
sys    0m0.000s
```

JIT has a speedup of 7x !

That’s all !

Thank you Folks !