Lua JIT and FFI

FFI

- A **foreign function interface** (**FFI**) is a mechanism by which a program written in one programming language can call routines or make use of services written in another
- The host language should be aware of **application binary interface (ABI)**, **calling conventions** and **runtime** of the guest language
- The example of FFI is **C++** which has trivial FFI with **C** enabled with extern “C” keyword

FFI with C

- Most of the high level languages has some sort of FFI with C: **Java JNA**, **.NET P/Invoke**, **Python CFFI**, etc.
- The **Lua JIT** implementation also has **FFI** module
- Usually when somebody refer to FFI they imply no **glue code** required in guest language. However formally **Java JNI**, **Python Ctypes** and **Lua C API** are also FFI

Examples of FFI

With glue and without it

```
extern "C" JNIEXPORT void JNICALL Java_ClassName_MethodName (JNIEnv *env, jobject
obj, jstring javaString)
{
    const char *nativeString = env->GetStringUTFChars(javaString, 0);
    //Do something with the nativeString
    env->ReleaseStringUTFChars(javaString, nativeString);
}
```

```
using System.Runtime.InteropServices;

public class Program
{
    [DllImport("user32.dll")]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, int options);

    public static void Main(string[] args)
    {
            method.MessageBox(IntPtr.Zero, "Command-line message box", "Attention!", 0);
    }
}
```

Why C and not C++?

- C ABI is simple and cross-platform
- C++ ABI is hard and implementation specific
  - Many **calling conventions**: cdecl, stdcall, fastcall, thiscall, vectorcall
  - The **sizes, layout** and **alignment** of basic data types and structures may vary
  - **Name mangling** is compiler specific
    - GCC - _Z1hi
    - MSVC - ?h@@YAXH@Z
  - **Exceptions** are compiler specific: Windows SHE, GCC zero cost exceptions, etc.
- Is it possible? Yes, but need to maintain code for different compilers

Lua JIT

C API

```
#include <math.h>
#include <stdio.h>
void h(int v)
{
    printf("Number of values: %d", v);
}
extern "C" int lua_sin(lua_State *L)
{
    double d = lua_tonumber(L, 1);
    lua_pushnumber(L, sin(d));
    return 1;
}
extern "C" int lua_h(lua_State *L)
{
    int d = (int)lua_tonumber(L, 1);
    h(d);
    return 0;
}
...
lua_pushcfunction(L, lua_sin);
lua_setglobal(L, "mysin");
lua_pushcfunction(L, lua_h);
lua_setglobal(L, "myh");
...
```

FFI

```
local ffi = require("ffi")

ffi.cdef[[
    double sin(double value);
    void h(int value) asm(?h@@YAXH@Z);
]]

local lib = ffi.load("mylibrary")

...
local result = lib.sin(1.57)
lib.h(result)
...
```

Meta Object Compiler

- LLVM + Clang based framework
- Produce reflection info and FFI definition code
- User decide what fields and methods to expose, and what additional info to attach
- Lua FFI struct with metatable behave as C++ class (as much close as possible)
- C++ classes can be derived in Lua
- Quite large subset of C++ is supported (potentially ☺)

Meta Object Compiler
class layout and member access

```
class Test
{
public:
    float value;

private:
    int m_count;
};
```

```
ffi.cdef[[
    typedef struct
    {
        float value;
        char __padding[4];
    } Test;
]]
```

Meta Object Compiler
virtual functions and inheritance

```
class Command
{
public:
    virtual double exec() = 0;
};
class Multiply
{
public:
    double exec(){return a * b;}
    double a;
    double b;
};
```

```
ffi.cdef[[
    typedef double (*Command_exec_ptr)(void *);
    typedef struct
    {
        Command_exec_ptr Command_exec;
    } Command_vft;
    typedef struct
    {
         Command_vft *Command_vftable;
    } Command;
    const Command_vft Command_vftable asm("??_7VCommand@@6B@");
    typedef struct
    {
        Command_vft *Command_vftable;
        double a;
        double b;
    } Multiply;
    double Command_exec(void *) asm("??_7Vexec@Command@@6B@");
]]
```

Meta Object Compiler
other

- Method overloading
  - Different names for C++ methods: Test_func_1, Test_func_2
  - Lua wrapper function with complicated logic which chose the closest method
  - Performance drop, C function cannot be called directly
- Named arguments
  - Lua wrapper function which accepts table with arguments as input
- Exception handling
  - Partially supported by Lua JIT
  - Disabled by default
  - Performance drop

Thank you