Cocoa from a Scripting Language

or "Lua-se the Compile"

Richard Kiss
him@richardkiss.com

Abstract

The dynamism of Objective-C makes it a good candidate for binding to scripting languages with a generalized gateway to the messaging system. We present here a discussion of issues related to making the Objective-C runtime available to a scripting language. The particular language chosen here is Lua, but much discussion can be generalized to the language of your choice, including gaining insight into issues facing the Java bridge.

Introduction

Many applications can benefit from scripting. Giving users access to a scripting language to control an application turns that application from a program into a platform, upon which users can build systems far beyond the imagination of the application author.

Nearly any application that runs unattended can benefit from scripting. For example, you may want to perform time-consuming computations (such as image editors, where users might want to perform a sequence of filters) or to serve remote users (such as web servers, IRC clients or file servers).

Adding scripting capabilities can be a challenge. First, you must create or choose a language. Although creating your own language may be an interesting project, chances are it is a distraction from the real problems your application is trying to solve. Once the language is chosen, you must then decide what objects or functions from your application need to be exposed to the scripting language, and you must provide bridge methods that convert the native types of your scripting language to the types expected by the exposed functions.

Lua is an excellent choice to add scripting capabilities to your application. Lua is powerful and easy to learn, but high-level enough to avoid the pointer problems that plague C. It was designed as an embeddable, extensible, dynamic language that can easily expose data structures and objects in your application.

If you use Objective-C, you may find the Lua bridge to be an excellent starting point. The bridge provides a way to call Objective-C from Lua and vice versa, with objects being transformed automatically as they cross the bridge, similar to the Java bridge.

The bridge provides Lua access to all Objective-C classes and most data types. You may find that it meets your scripting needs with no changes.

A Crash Course in Lua

Lua is a simple, yet remarkably powerful and complete language. Its syntax is sparse, and light on punctuation. The complete reserved word list is

```
and     break  do       else    elseif
end     for    function if      in
local   nil    not      or      repeat
return  then   until    while
```

The following strings denote other tokens, mostly operators:

```
 ~=  <=  >=  <   >   ==  =   +   -
 *   /   (   )   {   }   [   ]   ;
 ,   .   ..   ...
```

Two dashes ("--") mark a comment until end of line, similar to C++'s double slash notation ("//").

Although Lua is simple, it is too complex to

detail exhaustively here. We will provide some key examples to exhibit the flavour of the language, but for detailed information, consult the excellent reference manual at

<http://www.tecgraf.puc-rio.br/lua/manual/>.

Variables

Variables can be global (the default) or local. Variables are untyped: any variable can be assigned Lua values of any type. The six Lua types are: nil, number, string, function, table and userdata (which are roughly like C opaque types, in that they can only be created and operated on by C glue code). Table and userdata values can optionally have a "tag" subtype.

```
identity = nil
age = 30
username = "Gus" -- strings are 8-bit
data, null characters allowed, any
length
data = "\5\2\255" -- decimal values
pi = 3.14159265358979323 -- all numbers
are represented as C "double"
show = print -- "print" is initialized
to point to a built-in function; now
"show" points to it too
info = { flavour = "lemon", color =
"red", size = 200 } -- a table
```

Function Declarations:

```
function printGreeting(name, age)
  print ("Hello " .. name .. ", I
understand you are " .. age .. " years
old")
  return age+1
end
```

Invocation:

```
ageNextYear = printGreeting("Tom", 20)
pg = printGreeting
ageNextYear = pg("Tom", ageNextYear)
```

Tables:

Tables are associative arrays, with any non-nil Lua object as a key and any object as a value (including functions). This allows you to write simple object-style programming.

```
cousin = {
  age = 12,
  name = "Fred",
  greet = function() print "hi!" end
}
```

Tables can be modified at any time.

```
cousin.age = cousin.age + 1
-- birthday!
cousin.greet = function() print ("Well
howdy!") end
```

Table entries can be accessed using two distinct syntaxes.

```
cousin["age"] = 15
-- equivalent to "cousin.age = 15"
index = "age"
cousin[index] = 15
-- also equivalent to "cousin.age = 15"
```

All variables and table entries have the value "nil" unless otherwise set.

Conditional Statements (if-then):

```
if a ~= b then print("a is not equal to
b") end
-- use ~= as opposed to the C !=

if age < 13 then
  print("You are not yet a teenager")
elseif age < 20 then
  print("You are a teenager")
else
  print("You are no longer a teenager")
end
```

Looping Constructs (repeat-while-for):

```
a=0
repeat
  print(a)
  a = a + 1
until a >= 20
```

```
a=0
while a<20 do
  print(a)
  a = a+1
end

for index = 1,21,2 do
-- C version: "for(i=1;i<=21;i+=2)"
  print(index)
end

for index = 50,100 do print(index) end
-- default increment is 1
```

Use "break" to break out of repeat, while and for loops.

```
a=0
while 1 do
  print(a)
  a = a + 1
  if a == 15 then break end
end

-- iterating over an array
-- arrays are just tables, with n set
-- an array's first index is 1, not 0

array = {'a', 'b', 10; n=3}
for i = 1,array.n do
  print(i .. " -> " .. array[i])
end
```

Here is a function to dump a table:

```
-- iterating over a table

function printTable(table)
  for i,v in table do
    print("t['" .. i .. "'] = " .. v)
  end
end

table = { dog = 'Sparky', cat = 'Jane'}
printTable(table)
```

Lua idioms:

```
size = size or 10
-- equiv. to
--"if (size == nil) then size = 10 end"
```

Lua allows vector-style lvalues for multiple assignments.

```
x,y = y,x -- swap two values
a,b,c = b,c,a -- rotate three values
```

If the left and right side don't match counts, "nil" values fill excess.

```
a,b,c,d = 10,15
-- c and d each get the value "nil"
```

Functions can return multiple values.

```
-- an efficient implementation of
-- Fibonacci using recursion
function fib(n)
  if (n<=2) then return 1,1 end
  local fnminus1, fnminus2 = fib(n-1)
  return fnminus1 + fnminus2, fnminus1
end
```

Lua functions can take an arbitrary number of arguments:

```
function showall(...)
  local argCount = arg.n
  print("You called showall with " ..
argCount .. " arguments.")
  print("They are:")
  local index
  for index = 1, argCount do
    print(index .. " -> " ..
arg[index])
  end
end

showall("a","b",'c',100)
```

Functions can be return values. Functions as return values can "fix" values, yielding a function with fewer parameters.

```
function
functionThatPrintsString(string)
  return function() print(%string) end
end
```

The "%" means "fix this expression to a constant equal to the current value of the expression". So

```
h = functionThatPrintsString("hello")
print("h = ", h)
h()
```

yields the following output:

```
h =    function: 0x80682a0
hello
```

Object-style syntax

If obj is a table, then obj:met(p1,p2) is a shortcut for obj.met(obj,p1,p2). You can use this to do object-oriented style programming.

```
obj = {
      x = 10,
      y = 20,

      -- accessors for x & y
      getX = function(self) return
self.x end,
      getY = function(self) return
self.y end,
      setX = function(self, newx)
self.x = newx end,
      setY = function(self, newy)
self.y = newy end,

      -- move the x & y position of the
object
      moveTo = function(self, newx,
newy)
         self:setX(newx)
         self:setY(newy)
      end
}

obj:moveTo(100,200)
```

The Lua distribution comes with many fine examples of Lua programs ranging from the simple to the very complex. Take a look.

Extending Lua

Lua is a remarkably tiny yet complete language. However, it comes with only a few built-in libraries, and even these are optional. There is no built-in way to "break out" of Lua's little universe into the host application or operating system.

As an extension language, Lua has been designed to make it easy to call back to C. Here's an example that adds a new Lua function "system" that calls the C standard library system function.

```
#include "lauxlib.h"

void luaSystem(lua_State *L)
{
  const char *luaArg =
luaL_check_string(L,1); // get string
argument
  int returnValue;
  returnValue = system(luaArg);
  lua_pushnumber(L, returnValue);
}

void addSystem(lua_State *L)
{
  lua_register(L,"system",luaSystem);
}
```

Invoking "addSystem" will add the global function "system" to the Lua universe represented by L. Now

```
system("ls")
```

will work just as the C code would, returning the integer value returned by the C system call back to Lua. (Note that the Lua IO library already has a similar function named "execute".)

When Lua calls into C code, it passes in the current Lua state. The most important bit of Lua state is the "Lua stack", which contains the Lua values passed in. You can extract any of these values and operate on them, and you can return any number of values.

Using The Objective-C Runtime From Lua

Objective-C binds messages to implementations at runtime, not at compile-time or link-time. The most important function in the Objective-C runtime may very well be

```
OBJC_EXPORT id objc_msgSend(id self,
```

```
SEL op, ...);
```

defined in objc-runtime.h. (Note that the source for the Objective-C runtime is available as part of Darwin.) In fact, all messaging in Objective-C uses this function or one of its variants. For example, the Objective-C code

```
d = [n1 isEqualToNumber:n2];
```

actually internally looks more or less like the C code

```
d = (int)objc_msgSend(digits,
@selector(numberWithDouble:), n2);
```

The Objective-C runtime also makes the "method signatures" available, that is, the count and type of arguments the function needs, and the return value type. Using this information, we can coerce the Lua arguments to the correct Objective-C type. For example, Lua strings can be coerced to selectors or char pointers depending upon what is expected by Objective-C.

Providing a Lua glue function "objc_msgSend" opens up nearly all Objective-C runtime functions to Lua. This glue function does the following:

- look up the method signature for the given selector on the given instance
- for each Lua argument, coerce Lua type to the Objective-C type expected by the method signature
- invoke the method
- coerce the return value to something Lua can handle
- return to Lua

We use the Cocoa class NSInvocation which abstracts many of the nasty details of sending Objective-C messages and handling various return types.

With this single gateway function, Lua can now send nearly any message to any Objective-C object (as long as it can properly coerce the argument types).

But what if Lua needs to create an Objective-C object? Class methods act as object factories, so exposing Objective-C classes to Lua will do the job. It turns out that Objective-C classes are essentially singleton Objective-C objects.

Another Objective-C runtime function will return an Objective-C class by name:

```
OBJC_EXPORT id objc_getClass(const char
*name);
```

After we expose this function to Lua, we can then create Objective-C objects in Lua. For example,

```
nowDate =
objc_msgSend(objc_getClass("NSDate"),
"date")
```

The objc_getClass function returns the NSDate class, and the objc_msgSend functions sends it a "date" message, which returns a new NSDate. It's roughly equivalent to the Objective-C code

```
NSDate *nowDate = [NSDate date];
```

Yucky Syntax

Admittedly, the Lua syntax is pretty messy way of doing method invocations. Fortunately, Lua includes a powerful feature known as "tag methods", which allow tables and userdata objects of certain subtypes (or "tags") to override the meaning of operators. One such operator is the "gettable" operator, which is invoked when a table-lookup is performed. Therefore, if the Lua userdata object representing NSDate has the gettable tag overridden, we can rewrite the ugly

```
nowDate =
objc_msgSend(objc_getClass("NSDate"),
"date")
```

as

```
nowDate =
objc_getClass("NSDate").date()
```

Much more readable! Additionally, we can override the "getglobal" tag method -- invoked whenever a global variable is referenced -- to immediately expose all Objective-C classes simultaneously. This allows the following:

```
nowDate = NSDate.date()
```

Much nicer!

For more information on how this works, consult the Lua reference manual, section 4.8, "Tag Methods" and the source code for the Lua bridge.

Selector Name Translation

Objective-C selector names include colons and all keywords. For example, the class method of NSString

```
+ (id)stringWithCString:(const char
*)cString length:(unsigned)length;
```

has "stringWithCString:length:" as a selector name in the runtime.

Unfortunately, the ":" character cannot be used as part of an identifier type in Lua. However, "_" is permissible, so we translate "_" to ":" before looking up selector names. For example,

```
s =
NSString.stringWithCString_length_("new
string", 9)
```

Any arbitrary string (and in fact, any Lua object besides nil) can be an index in a table, so the following is equivalent (although strange-looking):

```
s =
NSString["stringWithCString:length:"]("
newstring", 9)
```

Now would be a good time to look at example 1, included with the Lua bridge. This example includes some simple messages sent to the Foundation framework from Lua.

Subclassing Objective-C Classes

Objective-C allows new classes to be registered at runtime. This is required to support loadable bundles, but it can be used to provide a sort of subclassing of Objective-C classes with method implementations in languages other than Objective-C. The WebScript scripting language of WebObjects 4 does exactly this.

A new Objective-C subclass has several properties, including: the superclass; new class methods; new instance methods; the class name; new instance variables.

The Lua bridge includes some code has been prepared that allows Objective-C subclasses to be created at runtime, with method implementations in Lua. These features are best considered experimental.

How to Use This Feature

A Lua subclass can contain new object methods and new instance variables. Both of these are encapsulated in a template table. Here's an example:

```
template = {
        age = 15,
        gender = "unknown",
        name = "",
        getName = function(self) return
self.name end,
        getAge = function(self) return
self.age end,
        setName_ = function(self,
newName) self.name = newName.cString()
end,
        setAge_ = function(self,
newAge) self.age = newAge.doubleValue()
end,
        addYearsToAge_ = function(self,
years) self.age = self.age +
years.doubleValue() end,
        description = function(self)
return
NSString.stringWithCString("Person
object: "..name) end,
}

subclass("Person", NSObject, template)
```

This example adds a new Objective-C class named "Person" to the Objective-C runtime, as a subclass of NSObject. Besides all the methods implemented by NSObject, it overrides

```
-(NSString*)description;
```

and implements these additional instance methods:

```
-(id)getName;
-(id)getAge;
-(id)setName:(id)newName;
-(id)setAge:(id)newAge;
-(id)addYearsToAge:(id)years;
```

For existing selectors, we use the superclass to get the method signature. For selectors not in the superclass, we count the parameters by counting occurrences of "_", and assume the return value and all parameters have type "id". That's why we must do conversions, like "newAge.doubleValue()" before operating on the values.

How Does it Work?

Creating a New Hybrid Subclass

The "subclass" call takes new class name, a base class, and a template. A new Objective-C class is created with a pointer to the Lua template table. Each instance has an Objective-C part and a Lua part, and each half has a reference to the other half. This requires a single extra instance variable to be added to the base Objective-C class. Hybrid classes have three methods are overridden by the Lua bridge:

```
-(id)init
-(void)dealloc
-(void)forwardInvocation:
(NSInvocation*)anInvocation
```

The first two method ensure that the Lua-side representation of the object is created and released. The last method is called by the Objective-C runtime when a message is sent to the Objective-C side of the object; it invokes the Lua code corresponding to the method.

The "subclass" method iterates over the Lua table. For each function, convert the name to a selector and see if it has a signature in the superclass. If so, use that signature; if not, use a signature with all ids, counting the parameters by counting "_" characters (there is no way to know how many arguments a Lua function is expecting). Selectors are registered with the Objective-C runtime if necessary.

For each method in the template, a stub entry is created to be added to the Objective-C method table. Each entry is a triplet of selector name, method signature, and implementation (see struct objc_method in objc-class.h). It's set up in such a way so that each method falls into the -forward:: method of NSObject, which encapsulates the parameters into an NSInvocation object and falls into

```
-(void)forwardInvocation:
(NSInvocation*)anInvocation
```

Invoking a Lua Method From Objective-C

Whenever the implementation of a hybrid object's method is in Lua, the Objective-C runtime forwards the call to the method -forwardInvocation:. This method is implemented in the Lua bridge.

This implementation examines the selector; fetches the corresponding Lua table for the instance (not the template); looks up the Lua function to call; transforms the Objective-C parameters into Lua types; calls the Lua function; transforms the return value and returns it.

Of course, there are a lot of details glossed over here. The code provides the best documentation for those interested.

Creating Hybrid Objects

The -init method of a hybrid object clones the template Lua table. The clone acts as the Lua-side representation of the hybrid object. The self.id table entry is set to point to the Objective-C half of the object.

Because of this, all -init... methods in Lua MUST invoke the original -init method to properly create the Lua-side representation. Otherwise invocations of Lua-implemented methods will silently fail because the instance variable pointing to the Lua object will be nil.

Neat Side Effects

If you're still with me, you may realize that nothing prevents the template table from being modified even after the subclass is created. In fact, nothing prevents the Lua table for an instance from being modified. Adding new functions to the table wouldn't do much (you would have to notify the Objective-C runtime that a new method was added), but you can change implementations for existing methods that are overridden. Since each hybrid object has its own Lua-side representation, you can change implementations on a PER-OBJECT basis! Very strange.

Why doesn't Objective-C provide this feature? Well, we could say that that's just the way it works. But there are speed penalties to each object having its own dispatch table. Objective-C strike a balance between dynamism and flexibility by having a class-level cache of selectors and implementations. For example, the first time -init is called on an NSString object, the runtime must perform a time-consuming look-up of selector to implementation. However, it then caches the implementation in the NSString selector cache, so subsequent calls of -init on an NSString object skip the look-up step.

Of course, another answer is that there's not really any (easy) way to add Objective-C code at runtime, since compilation creates a file, not a RAM image. Technically, one could create an object file and link to it, but that's a lot of work (and obviously processor dependent).

See the Lua bridge for an example that illustrates method implementations that differ on a per-object basis.

Strategies to Add Scripting

One thing for certain: end user code must be invoked for scripting to be useful. To do this, simply call the C function "dobuffer" in the Lua library on a buffer containing the user's code. It will compile the code (if necessary) and execute it. The question becomes, when is the user's code executed, and what does it look like?

File-Based Strategy

With this strategy, each event that requires scripting invocation is associated with a separate file. When an event occurs, it scripts the file.

Advantages: easy to document (when X happens, the file "Y.lua" is executed). Easy for user the change features, even while application is running (simply change the Lua file).

Disadvantages: one file per event. No input values (besides globals). No output values (besides globals).

Function-Based Strategy

With this strategy, each event is associated with a specific function name in the global name space. For example, the Lua function "fileDownloadRequest" could be called whenever a file download request is received, and you could pass in event-specific parameters such as file path, user information, etc. You need to invoke a file at start-up time that defines the function names in question.

A problem is, how dynamic is this? If the user wants to change the behaviour of an event, she might edit the file, then be puzzled as to why the changes aren't taking effect (since only the assigning of function names actually affects the change). A possibility is to cache the last modified date of the start-up file, and re-execute it whenever it is "new enough". The downside to this is that all top-level code -- not just function assignments -- will get re-executed too.

Advantages: all events can be in one file. You can provide reasonable default functions. Can pass in arguments and use return values.

Disadvantages: possibly less dynamic.

Notification-Based Strategy

With this strategy, you document notifications, and leave it up to the user to register for the NSNotification types she is interested in. This strategy may fit in well with your existing

notification model. One problem is that notifications are reported only after an event occurs, and the notification recipients therefore cannot prevent the event from taking effect.

Advantages: closely tied to Objective-C runtime and Cocoa.

Disadvantages: closely tied to Objective-C runtime and Cocoa. Notifications cannot return values.

The Lua bridge project has examples that illustrate each of these strategies.

Conclusion

The dynamism of Objective-C makes it easy to bind large portions of the Cocoa API to a scripting language with just a few generalized gateway functions. The dynamism of Lua allows creation of hybrid classes created at runtime, with method implementations written in Lua. The combination provides a very general framework that can meet application dynamic scripting needs with little work on the part of the application author.

Recommended Reading

[Lua Web Site]
<http://www.tecgraf.puc-rio.br/lua>

[Objective-C Documentation]
<http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/index.html>

[Foundation]
<http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/index.html>
In particular, NSMethodSignature and NSInvocation.

[Darwin]
<http://www.publicsource.apple.com/>
<http://www.publicsource.apple.com/projects/darwin/1.2/projects.html>
Source code for Objective-C runtime is in objc4 project.

[GNUStep] <http://www.gnustep.org/>
Includes source for Foundation-compatible framework, which can provide insight into implementations.