Traveling Light, the Lua Way

Ashwin Hirschi, *Reflexis*

Lua can help you become more productive by extending your C/C++ creations with the expressive power and flexibility of a dynamically typed language.

Five years ago, our team at Reflexis ran into a little language from Brazil. Lua (pronounced loo-ah) changed the way we work profoundly. It lets us create hybrid solutions that combine the strengths of statically typed software with the flexibility of a dynamically typed environment. In short, with Lua, we get the best of both worlds.

We've used Lua to create software ranging from small in-house tools, to highly dynamic desktop systems, to a browser-based platform for designing and deploying complex Web applications. Lua extended the range of solutions we offer. But it also helped us drastically reduce the time and effort needed to finish our projects.

In this article, I'll share some of our experiences, a few of the lessons we've learned, and talk about changes we've made to our systems and the way we develop solutions.

Getting started

Starting a software company is a bit like setting out on a long journey. So, when our small band of developers at Reflexis headed out, we did our homework. For this trip, we knew what we wanted to do and packed accordingly.

Unsurprisingly, a few years down the road, reality kicked in. For one thing, being small means bigger fish are swimming in the pond. Bigger companies tend to have bigger budgets. Often these budgets translate into much louder shouting when it comes to attracting customers to your services.

Once you find yourself pondering how to keep paying the rent, several options present themselves. Panicking might be the natural choice. Rethinking your business model is a good alternative (especially if you're bent on surviving or the panicking starts to wear you down).

But our situation drove home another lesson as well. It taught us to not waste energy and to make sure we get the most out of the time we spend creating our software. And at the risk of stating the obvious, here are some of the reasons.

First, the more easily you can put together an exciting demo, the more interested a possible client will be. Getting prospective customers really interested is good; they'll listen longer to whatever you want to tell them.

Second, the less development time you spend on juggling technical details, the more effort you can pour into truly fitting a solution to your customer's needs. A fully satisfied customer is a fine thing to have; they tend to tell their friends.

Finally, the faster you can deliver whatever solution you want to develop, the sooner

0740-7459/07/$25.00 © 2007 IEEE

Lua is an elegant, easy-to-learn language with a mostly procedural syntax.

you can move on to your next project. You could charge your client less or simply earn your pay more effectively. Having choices like these is nice.

Now, if outshouting the competition isn't your cup of tea, why not try to put some distance between them by becoming quicker on your feet? Think of rapid application development as a way to survive. The *accelerate out of problems approach* (AooP?), if you will.

So, at this juncture, we took a long, hard look not only at our business model but also—and perhaps more important—at how we build our software. Our expertise centered on C++/C and a handful of dynamically typed languages such as Tcl, Python, and Perl. These were the tools of our trade. They served us well and let us deliver desktop and Web applications.

However, despite having used many of these tools for more than a decade, we felt that they didn't help us move as quickly as we wanted to. Progressing rapidly in C++ almost felt like a contradiction in terms. Creating scripts to flesh out an idea often meant feeling frustrated when we had to redo everything for the actual deliverable.

But while some might think that the grass is much greener on the other side, be it a platform such as Java or .NET, we were happy with our products and libraries in C and C++. The software we used, mostly homegrown or open source, worked well. Why drop everything and leave for pastures unknown? Perhaps the only thing missing was something to bridge the gap between the advantages of our existing C/C++ systems and the pleasures of doodling in a more dynamic environment.

The answer to our predicament came in the form of a little language from Brazil. Portuguese for "moon," Lua is a portable, lightweight, and dynamically typed scripting language that wraps a wealth of state-of-the-art mechanisms in an elegant syntax. Designed for extending applications, it also sports an intuitive C API, making integration into existing software extremely easy.

As a direct result of adopting Lua, we left behind a lot of technical baggage that we'd deemed essential when we set out. In its place is something much lighter and more powerful, making our working lives more efficient, and the road much more pleasant to travel. I hope to show that as long as you're in the business of writing software, less can indeed be much more if you do it the Lua way.

First steps

Our initial steps with Lua were quick and painless. Lua comes in the shape of an open source library written in clean ANSI C. It compiles out-of-the-box on many different platforms and operating systems. The distribution includes a concise, well-written reference manual that details both the language and the C API exposed by the library.

An excellent way to get to know Lua in depth is by reading *Programming in Lua*,[1] by Roberto Ierusalimschy (chief architect of the language and part of the Lua team). Although a first edition of the book is available online, a newer edition describes the latest features and adds a substantial amount of useful examples and new material.

After reading the book (or manual[2]), a good way to come to grips with Lua is to write a few simple scripts and try them out using the interpreter (included in the distribution). Any errors, either during compilation or execution, will prompt the interpreter to display an alert, consisting of a helpful message and a call stack indicating where the mishap occurred.

I won't discuss particular language details here. Those are best left for another, more technical article but can also easily be gleaned from other sources, such as the official Lua site (www.lua.org). Suffice it to say that Lua is an elegant, easy-to-learn language with a mostly procedural syntax, featuring automatic memory management, full lexical scoping, closures, iterators, coroutines, proper tail calls, and extremely practical data-handling using associative arrays.

Although Lua isn't an object-oriented language, it provides powerful metamechanisms that let you implement classes and inheritance. But while these metamechanisms let you change the language's semantics in all sort of unorthodox ways, Lua's extensibility goes much further: from the ground up, Lua was designed to extend other applications. So, if you really want to grasp Lua's full potential, you need to look into *bindings*.

Bindings

A binding is basically a layer of glue code that sits between Lua and another piece of software. For instance, creating a binding to a C/C++ library exposes that library's functionality to the Lua environment. This lets you use the library in Lua scripts without needing to compile and link C programs.

```
/* Holds the handle to the device context of the active window */
static HDC current_dc;

/* A simple wrapper to set or get text color output */
int rfx_textcolor(lua_State *L)
{
 int old_color;

 /* Was a number passed in? */
 if (lua_isnumber(L, 1))
  old_color = SetTextColor(current_dc, lua_tonumber(L, 1));
 else
  old_color = GetTextColor(current_dc);

 /* Pass back a single (number) result */
 lua_pushnumber(L, old_color);
 return 1;
}

/* How to register this function in your Lua state */
lua_register(L, "textcolor", rfx_textcolor);
```

Figure 1. A binding that exposes Windows Graphics Device Interface functions to get or set the color of text output by adding one function to the Lua environment.

You can create bindings in numerous ways, such as writing them by hand or using tools (see http://lua-users.org/wiki) to generate them from header files. I prefer the former because it allows for much greater control of how functionality is exposed to scripts. This makes it possible to come up with a higher-level interface than the original library might provide, resulting in a system that's far more convenient to use.

Figure 1 shows an example of a tiny binding, exposing Windows Graphics Device Interface functions to get or set the color of text output by adding one function to the Lua environment.

The first Lua binding I wrote was for an XML handling library I'd developed a few years earlier. The library ended up in several of our products, and because it had a straightforward C++ interface, I thought it would make a good candidate for my little experiment. Somewhat to my surprise, I came up with a full, working binding in just a handful of hours, courtesy of the well-documented, stack-based API Lua exposes to (binding) programmers. The surprise was all the more pleasant because it didn't take long for us to realize we could now use our library by simply composing scripts in a text editor and running them through the Lua interpreter.

Reading XML data, manipulating the structure, and producing some relevant output had become a mere matter of writing concise Lua code. Not only did the endless write/compile/link/debug sessions in top-heavy C/C++ IDEs start to feel like a complete waste of time, but our existing products, with their fixed, hard-coded C++ data manipulations, suddenly seemed wholly old-fashioned! Needless to say, my little Lua and XML experiment proved very encouraging and helped us decide to invest more time and energy in Lua.

Embedding

As the previous example has shown, it can be worth your while to integrate Lua with your libraries. Embedding is another way of using Lua to your advantage. Instead of using the basic (command line) interpreter to access your libraries, you can enhance an entire system, such as an existing product, by incorporating Lua into it. Once you've embedded Lua, its host program gains a mature, powerful scripting facility and can create and control as many Lua virtual machines as required. You can load both data or code into any virtual machine (VM) at any time. The result is flexible, fine-grained control over what happens where and when.

Possibly the most basic scenario is simply using Lua to maintain system configuration data. Configuration data can mean many

You can use dynamic Lua scripting to complement your existing statically typed systems.

things, ranging from user settings and interface definitions to small structured databases and raw application data. Handling data through Lua is especially convenient because its tables (think efficient, associative arrays) are a clean but powerful data description mechanism. Once you've structured the information correctly, actually loading data structures and accessing them is trivial and fast.

In a second, more elaborate usage scenario, you might embed Lua to provide hooks for behavior. For instance, you can integrate Lua into your product to give end users the means to script or change your application. The product's developers decide what can be scripted because they control which product functions are available to the scripting environment.

As an interesting variation on this scenario, you might use dynamic scripting to define and drive parts of the base product. Once you expose sufficient functions to the Lua scripting side, there's often little need to keep an application's behavior on the statically typed side of the implementation fence.

In fact, moving product behavior to the Lua side offers manifold advantages: you can develop code more easily, load logic on demand, set up tests faster, and identify and catch bugs sooner. Additionally, behavior becomes more accessible and thus more convenient to review, and the overall product is easier to evolve.

Although many of the points I've mentioned might also apply to developing in other dynamic scripting languages, Lua distinguishes itself by letting you actually mix and match your existing C/C++ technology with the qualities of dynamic scripting as (and when) you see fit. In short, you keep the strengths of (and investments in) your existing software while Lua opens a pathway to reaping the benefits of a dynamically typed language at your leisure.

Desktop solutions

At this point, it's clear that you can use dynamic Lua scripting to complement your existing statically typed systems. There are many ways to create these "hybrid solutions" in an elegant, effective manner. But sometimes, the proof of the pudding is in the producing of useful products. So, after a few weeks of initial investigation, our team decided to take the plunge and incorporate Lua into our next big project, a client-server-based information management system.

A key component for this new system was a lightweight, dynamic front end. We wanted something that could act as a richer client than browsers could, with features such as excellent keyboard navigation, context menu support, and good state management. Because the system had to be readily available to a potentially large number of end users, we also needed to ensure this front end required no installation but would be small enough to run off our client's network.

Lua to the rescue

Fortunately, in between actual work, we'd been experimenting with compact desktop runtimes. We were able to create applications with tiny footprints by avoiding the usual route of big and bulky GUI libraries and pretty much doing everything ourselves instead. This approach works surprisingly well but shares an important downside with other, more traditional techniques: the static nature of the implementation language doesn't really lend itself well to creating flexible user interfaces. In other words, this looked like the perfect opportunity for Lua to come to the rescue!

Because we didn't have an existing product to integrate Lua into, we started with a nice clean slate. So, we chose to develop a generic Lua-driven runtime that could support a wide range of GUI applications. Perhaps the best way to picture this "platform" is as a GUI alternative to the standard interpreter, albeit with a lot of extra goodies built in.

We started by integrating Lua with the minimalistic technology prototyped for our tiny footprint runtime. Because Lua is itself very small (the library weighs in at approximately 100 Kbytes), it added relatively little to the code size. Soon, we had something up and running, with Lua code able to draw on a window canvas.

Because GUI applications are mostly event driven, the next step was wiring windows message events to handlers registered (that is, functions loaded) in a Lua VM. For instance, if the user moves the mouse or presses a key, the runtime is notified and tries to look up the Lua function that should handle that event. If the function is present, the runtime calls it, effectively dispatching the message. If the runtime can't find a handler, it simply provides default behavior (usually a "no operation").

Odd though this might seem, we got things off the ground amazingly well with this simple

setup. Given that Lua scripts have a decent set of graphical (and textual) primitives available to them, you can comfortably control the entire look and feel of an application hosted in such a runtime through dynamic scripting.

At this point, we also noticed that Lua is fast—very fast. Initially I had doubts about, for example, being able to drive the rendering of the entire user interface. But these doubts evaporated when we found that we could smoothly render even complex GUIs through Lua logic while resizing application windows on our ancient (pre-2000) laptops.

Final touches

Obviously, a devil or two always lurks in the details. So I won't hide the fact that a lot of water passed under a nearby bridge before we were reasonably satisfied with how, for instance, controls and widgets looked and behaved. But while the runtime's open-ended, completely scripted GUI system started out with some rough edges, Lua's dynamic nature ensured that we could always implement new insights at a moment's notice. And because Lua has its own integrated compiler, we never needed elaborate IDE-hosted edit/compile/link cycles. Although you can precompile scripts, this is entirely optional: a Lua VM will quickly parse loaded, noncompiled code.

In short, once you're set up, all you really need is a text editor, a clear mind, and a save button. Restarting the application runtime or, better yet, simply keeping it running and reloading the Lua logic (using a reserved command key) does the rest and was enough to check our changes and move on. Furthermore, because everything was soft-scripted, we came up with all sorts of neat features that we probably couldn't have easily added otherwise.

For example, you can load user interface definitions on demand and automatically size and position controls. Listviews can contain any type as well as any (large) number of items. Adding new types of controls is easy. Font usage adapts to available font files, and you can easily resize texts at runtime (to accommodate different resolution settings and so on).

One of the more unconventional features we added is a *booklets* mechanism. Basically, it lets us run several different GUI "applets" in the same runtime. So, we can split larger applications into logical parts, which maintain their own user interface, letting users switch between them without losing state in any of the subsystems.

To ensure a smooth user experience, we added support for multithreading as one of the final touches. This way, operations that take longer to complete, such as exchanging data with a remote back end, don't force users to stare at frozen screens. Because we wanted an easy-to-program threading model, we decided to hand each thread its own VM. That way Lua logic can run in parallel without needing additional orchestration, because Lua VMs are completely isolated from each other.

Thread-safe, asynchronous message queues handle information exchange between code in different VMs, using a simple event-driven paradigm to complete the picture. This threading scheme worked out so nicely that even though we developed this Lua-driven runtime for our client-server front end, we ended up using it as the platform for the application server as well.

I have a lot more I could tell about our platform, especially because Lua makes programming such a pleasure. But for now, I'll just say that we've been using it for several years as the basis for all our desktop solutions, and I simply can't imagine ever going back. The details of this platform aren't the point here, though. What's important is that Lua lets you create hybrid solutions like these by implementing bits in statically checked systems where necessary and by putting pieces in dynamic scripting where possible.

Rapid development

As you can see, Lua plays a large part in how we produce software. We've talked about the lightweight Lua-driven platform we developed for desktop solutions, but make no mistake—Lua is just as well suited to developing for the Web.

Lua's small footprint and excellent performance make it ideal as a scripting language for dynamic Web sites. In fact, we found Lua so efficient that we deploy many of our Web applications using a basic CGI (Common Gateway Interface) setup.

Incorporating Lua into both our desktop and Web development had interesting advantages too. For starters, we can share and reuse script libraries, bindings, and tooling across these platforms. As an example, the SQLite (www.sqlite.org) binding that we originally developed to drive Web applications also ended up in our desktop runtime.

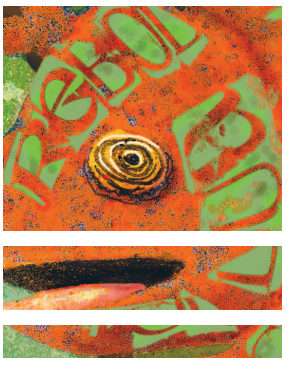

Figure 2. A Lua snippet using an SQLite binding.

```
-- how to "evaluate" select statements...
function sql.eval(db, query, fun, ctx)
 if type(fun) ~= 'function' then return nil, 'no fun' end
 if type(ctx) ~= 'table' then ctx = {} end

 local sm, err = db:prepare(query)
 if not sm then return nil, err end

 while 1 do
   local row = sm:next()
   if not row or fun(row, ctx) then break end
 end

 sm:finalize()

 return ctx
end

-- open a database and print & collect data
local db = sqlite "accounts.db"

if db then
 local persons = sql.eval(db, "select id, name, birthday from
 person;", function(row, ctx)
   local name, birthday = row[2], row[3]
   print("person: ", name, "birthday:", birthday)
   ctx[row[1]] = { name = name, birthday = birthday }
 end)

 -- persons now holds a mapping from IDs to names & birthdays

 db:close()
end
```

Figure 2 shows a somewhat simplified Lua snippet using our SQLite binding (for a more detailed version, see www.reflexis.com/ieee). The `prepare` and `finalize` calls are wired almost directly to their SQLite counterparts. But the `next` method wraps a SQLite step operation and several subsequent column access calls. We could have implemented the sql.eval function in the binding C code. But having it on the Lua side makes it easier to experiment with its behavior.

But it gets better. Our Lua-based Web runtime has a lot in common with its desktop cousin. So, we can deploy any Web application we develop offline, by simply hosting its logic in our desktop runtime and adding a small (Lua scripted) HTTP server to establish connections with browsers.

All we need to deploy these portable Web applications, or indeed our regular desktop solutions, is one compact (220 Kbytes) runtime executable plus an application package. We implement these packages using an archive (ZIP) or database (SQLite) format. So, we can easily bundle an application's Lua logic and data into single files. For deployment, no install is needed: just copy both the runtime and package and run.

If there's a secret ingredient here, it's simplicity. Calling simplicity a secret might sound paradoxical. But it often seems the software industry is trying to kill complexity by clubbing it over the head with more complexity (thus completely defeating the purpose).

Unfortunately, many things don't start out simple, and it takes serious effort to make them so. Goethe wasn't kidding when he wrote "In der Beschränkung zeigt sich erst der Meister." Loosely translated, that means "It's by restriction that the master shows himself" (also known as "Less is more").

A similar philosophy, called *economy of concepts*, runs throughout Lua's history. Instead of hardwiring all kinds of features into the language, the Lua team often designed metamechanisms to let developers program the features they require.[3]

In this respect, Lua can be a bit misleading. Although its straightforward syntax makes it easy to get into for beginner and expert programmers alike, it takes a while for people to grasp the possibilities hidden beneath its easy-going exterior.

As an example, one of Lua's features I treasure most is allowing the use of functions as first-class values. Combine them with the elegant syntax and available metamechanisms, and you can blend a procedural style with functional-programming techniques and object-oriented structuring. It's not something many people pick up on first contact. But once they do, it becomes quite hard to let go.

You also shouldn't underestimate the effect of Lua's ease of extensibility. Once you have bindings in place and a hybrid system starts to take shape, Lua becomes the glue between different pieces of software. This triggers an interesting dynamic: you realize you need to find a balance and decide what code goes where.

We found that with the right balance, the whole indeed becomes more than the sum of its parts. Scripting makes the pieces of the puzzle more accessible but also lets you quickly make parts interact in new and interesting ways. And so, Lua helps you simplify your solutions.

And this brings me to my final point. Lua helps you simplify how you work. It helps do away with the many technical details that plague developers but are often of little actual concern to the solutions they want to build. Removing this noise from the development process results in a newfound focus. And with this focus comes the ability to observe, think, and learn while you develop.

In the end, this ability to learn while you develop is the true key to rapid application development.

Smooth sailing

Driven to optimizing our development to be more competitive, our team stumbled upon Lua. Like many other dynamically typed languages, Lua provides a convenient syntax and a comfortable, forgiving runtime environment. But it was Lua's simplicity, speed, small footprint, and extensibility that impressed us most. Working with Lua has taught us several things:

Lua's ease of integration showed us that static and dynamic type checking aren't mutually exclusive. Together, they can become a killer combo. Better still, together they let you develop rapidly without dropping your previous efforts and investing in an entirely new platform.

We find ourselves scripting in Lua wherever possible and only coding statically (in C/C++) when occasionally needed. With more logic on the Lua side, we find our systems more flexible and much easier to evolve.

When doubting how to strike the right balance, it's best to benchmark and see for yourself. We built a high-resolution profiler into our desktop runtime. It lets us monitor and benchmark real-world solutions. But as a testament to Lua's speed, I haven't used the profiler for several years now.

Developing rapidly is one thing, but how about rapid deployment? Again, Lua helped us out. While its small code size helped us devise our compact desktop runtime, we use Lua's dynamic nature to load and run both data and script logic from single application packages on demand.

Application packages might seem like a silly technical detail. But we've found our clients very appreciative (as well as a little stunned) when product updates come in tiny, single files that they can simply download and save to their machines.

Sometimes people familiar with our hybrid approach ask me about the downsides to using Lua. Surely, traveling with Lua can't have been smooth sailing all the way? I have a hard time answering them. The simple fact is, we haven't had any difficulties.

When we started to use Lua seriously back in 2002, the language lacked support for weak references. But because Lua is open source and well written, we easily added the support (and weak tables have since been incorporated into Lua 5).

The biggest bump we encountered as rela-

Lua's ease of integration showed us that static and dynamic type checking aren't mutually exclusive.

About the Author

Ashwin Hirschi founded Reflexis, an independent group of software developers and consultants that provides R&D services around the globe. His interests include rapid development and deployment, task-based systems, knowledge management, and generally finding better ways to match information technology to people's needs. Contact him at ahirschi@reflexis.com.

tively early adopters were the changes the Lua team made when releasing Lua 5. Although the language gained many interesting new features, the underlying changes to the C API meant that we needed to rewrite our runtimes and bindings for compatibility.

Fortunately, this doesn't really affect anyone getting on the Lua train today. Also, once you've incorporated Lua into your systems, no one is forcing you to track new developments. In fact, many of our products still use Lua 4 because there's simply no need to switch.

One thing newcomers should consider is that Lua doesn't (yet) come in a "batteries included" flavor. Compared to other dynamically typed languages such as Python and Tcl, which feature extensive library support, the Lua distribution is decidedly bare bones. So, yes, some assembly might be required.

The Lua team has intentionally kept things clean and simple by releasing Lua mostly as a language library. Other initiatives, such as the Kepler project (www.keplerproject.org) and LuaForge (http://luaforge.net), focus more on making binaries, tools, bindings, and add-ons available. Anyone looking for a more fleshed-out environment would do well to check them out.

Lua offers all the benefits of dynamic scripting in a small, speedy package. If you're living in a C/C++ world, as we did way back when, its elegant extensibility lets you evolve the fruit of your labors by adding dynamic features if and when you need them.

Lua has helped us transform our products and reorganize our libraries and tools, stripping out the pointless, technical overhead and making them considerably more accessible and lightweight. Freed from the need for IDEs or high-end machines, we can work wherever we go. And without any noise to distract, we find it much easier to concentrate on building the solutions we want to create.

Let's not forget, people make software. So when you're developing solutions, what you want most is a clear state of mind. Lua has helped us get there, and I sincerely hope this story will point the way for others as well.

In the meantime, I'm looking forward to the road ahead. Life with Lua is good.

References

1. R. Ierusalimschy, *Programming in Lua,* 2nd ed., Lua.org, 2006.
2. R. Ierusalimschy, L.H. de Figueiredo, and W. Celes, *Lua 5.1 Reference Manual*, Lua.org, 2006.
3. R. Ierusalimschy, L.H. de Figueiredo, and W. Celes, "The Evolution of Lua"; www.tecgraf.puc-rio.br/~lhf/ftp/doc/hopl.pdf.