

Fully automatic Lua binding for C based on compiler AST

(Automatyczne generowanie bindingu z języka C do Lua na podstawie
drzewa AST)

Mateusz Łuczyński

Praca licencjacka

Promotor: dr Piotr Witkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

1 maja 2025

Abstract

Lua is a light-weight, fast and portable scripting language. Because of it, many applications choose to embed it's interpreter inside them, allowing the developers to extend their work without the need of rebuilding the core source code. In large projects this can save a lot of time spent on recompilation, during which the application can be rendered unusable. A well written official API and Lua's close connection to the C language, make C/C++ applications a perfect subject for the aforementioned process. We can see some examples of it in popular video games, like *World of Warcraft* or *Roblox*, where the ability for the dedicated fanbase to extend their favorite game takes a huge role in the projects success. In my work, I implemented a tool for automatically generating C code required for running Lua scripts, which can call API functions (written in C) that are part of the application. That way, given the source code containing the API calls definitions, we can add the extension capabilities to the project with little to none extra work.

Streszczenie

Lua to niedużych rozmiarów, szybki i przenośny język skryptowy. Te cechy sprawiają, że dużo aplikacji jest kompilowane razem z jego interpreterem, co pozwala deweloperom na modyfikacje funkcjonalności bez konieczności przebudowywania głównego kodu źródłowego. W dużych projektach prowadzi to do istotnej oszczędności czasu spędzonego na rekompilacji, podczas której aplikacja może nie być zdatna do użytku. Dobrze udokumentowane API oraz bliskie powiązanie Lua z językiem C, powoduje że projekty napisane w C/C++ idealnie nadają się do wykorzystania wspomnianego podejścia. Przykładami mogą być popularne gry wideo takie jak *World of Warcraft*, czy *Roblox*, w których możliwość rozwoju gry przez jej fanów jest ich sporą częścią i w dużej mierze przyczyniło się do ich sukcesu. W mojej pracy podjąłem się zaimplementowania narzędzia służącego do automatycznego generowania kodu w C, potrzebnego do uruchamiania skryptów w Lua, które korzystają z napisanych w C wywołań bibliotecznych będących częścią aplikacji. W ten sposób, mając kod źródłowy zawierający implementacje API, można bez dodatkowej pracy dodać możliwość dynamicznego rozbudowywania do swojego projektu.

Contents

1	Introduction	7
1.1	Project background	7
1.2	Motivation	8
2	Lua C API overview	9
2.1	Core concepts	9
2.2	Calling C functions from Lua scripts	9
2.3	Obstacles and limitations	9
3	AST generated by the compiler	11
3.1	General structure	11
3.2	Function arguments and their types	12
3.3	Handling user defined data structures	13
3.4	Function pointers	15
3.5	The output format	15
4	Generating the binding code	17
4.1	Fetching the arguments	17
4.1.1	Simple types	17
4.1.2	C structs and unions	17
4.1.3	Callbacks and closures	17
4.2	Error handling	17
4.3	Exposing the interface	17
5	Example usage	19

6	Conclusion	21
6.1	Possible further development	21

Chapter 1

Introduction

Embedding Lua into applications is a common practice among many projects which support dynamic expansion or configuration of some of its components. The language itself is extremely simple (only twenty keywords) but powerful at the same time, the official website claiming it to be the leading scripting language in the video game industry.

This thesis will focus on integrating Lua into projects written in C, specifically on exposing API calls to Lua scripts and running such scripts from within the main application. To do so, we need to write appropriate wrappers (we will call a collection of those wrappers the *binding* code) for each exposed API function, based on its signature. Fortunately, everything needed can be found within the *Abstract Syntax Tree* generated by the compiler.

The project consists of two main parts: the AST parser and the binding generator. Both are written in pure Python, with the parser producing an intermediate JSON file. Further analysis will be based on the GCC compiler, but with the parser being an isolated component, there is a possibility of writing an auxiliary one for a different tool.

1.1 Project background

The project originated as part of *Innovative projects by Nokia* initiative, where it was limit-tested and prototyped by a team of three students (including me). I was responsible for the code generation part of it, with my teammates developing unit tests and the AST parser. As part of this thesis, and with appropriate acknowledgement from my team, I rewrote and heavily expanded the code base, focusing on taking it out of the prototype form and turning it into a complete project.

1.2 Motivation

During the prototyping phase we focused on realizing a concrete scenario: let's say we have an application (which we will call the *scheduler*) that stores callbacks sent to it, and calls them when an appropriate event occurs. The main part of the *scheduler's* interface is the *register* function, that accepts a pointer to a function. Now, we would like to be able to pass callbacks written in Lua to the *register* function, and be able to call them even long after the script has finished registering them. That way, the *scheduler's* reactions for different events can be easily swapped and modified. At the same time, we would like the Lua components to be able to use functionality available to standard callbacks passed from within the application code, like a series of curated API calls using even user-defined C data structures.

Chapter 2

Lua C API overview

2.1 Core concepts

2.2 Calling C functions from Lua scripts

2.3 Obstacles and limitations

Chapter 3

AST generated by the compiler

The GCC compiler provides a set of options that allow developers to inspect different parts of the compilation process. Among these options are some that produce debug dumps from various points in the compilation, print some statistics like memory usage and execution time and list specific information about the compiler configuration (such as where it looks for libraries). Our main interest lays within those that allow to look into the process of creating the *Abstract Syntax Tree*. The one chosen for the project is called `-fdump-tree-original-raw` — it tells GCC to produce a file containing the intermediate language tree from the *original* pass, in its *raw* form (unlike the default, C-like representation). The produced dump is in an easily parsable format and contains every information we need:

- API function signatures — name, return type and argument types
- For user defined data structures — their names and fields they contain
- For function pointers — essentially the same information we need about the API functions

To know how we can extract this information we first inspect the tree representation in more detail.

3.1 General structure

The whole AST dump consists of *chunks* (designated by a line beginning with `;; Function`, followed by the relevant function name), which in turn consist of a series of *nodes*. Each node begins with an at sign (`@`) and an unique index number (along with a type of the node), and contains several key-value pairs, where each value either links to another node or represents some kind of property of the current one. Even though the format is fairly simple, the documentation is non-existent or really

hard to find — fortunately a set of experiments allows us to learn more about each node and its relation to our task.

For now let's assume we are working with a single function called `foo`, which takes an `int` as an argument and returns one as well. After invoking the compiler with the previously mentioned option, we should get a dump file containing a single chunk of nodes. The number of nodes will obviously depend on the implementation of `foo`, but some of them will always exist and they are the most useful ones. Our main interest will be focused on the `function_decl` node — its a central part of every node related to the API call signature.

Listing 3.1: Example `function_decl` node

```
@14  function_decl  name: @20  type: @21  srcp: api.c:5
                        args: @15  link: extern
```

From this node we can branch out in different directions: follow the node chain starting with `name` to get the function name, get information about the return type from the `type` node and inspect function arguments using the `args` node. Other nodes that we should pay attention to are `parm_decl` for function arguments, `field_decl` for struct definitions and various nodes ending with `_type` — these and more will be covered in more detail in the relevant sections.

3.2 Function arguments and their types

Following our `foo` example, let's first look at the way to extract function arguments. As mentioned before, we can follow the `args` property of the `function_decl` node to gather information about function arguments. This will show that they are stored in nodes of type `parm_decl`.

Listing 3.2: Example `parm_decl` node

```
@15  parm_decl  name: @21  type: @22  scpe: @14
                        srcp: api.c:4  argt: @22
                        size: @11  algn: 8  used: 1
```

As we can see from the example, the node conveniently points to everything we need — mainly the type, but the name is also there. There's one problem however — there seems to be no way of getting more than one argument this way. More specifically, there is no property linking to the rest of the arguments. We could try finding every node marked as `parm_decl` listed in currently processed chunk, but this creates another, more serious problem — we have no way of determining the correct order of arguments. This is especially crucial because without establishing the order we have no way of calling the API later, during the generation of the binding code.

The solution is not too far off though. Looking at the node pointed to by the `type` property of the `function_decl` node, followed by the `prms` property of the resulting node we end up with yet another node type: `tree_list`.

Listing 3.3: Example `tree_list` node

```
@28  tree_list  valu: @22 chan: @30
```

This is by far the simplest one, containing only two properties. It turns out that the `valu` property points exactly where we want to end up — that is the `_type` node corresponding to a function argument (described in a moment). On the other hand, the `chan` property, if present, points to the next `tree_list` in the chain. This way, we can extract all argument types in the correct order. There is a small caveat to this approach though, as we cannot easily get the name of the currently processed argument. This is a price we can pay though, as the names in this case are purely optional and play no role in the binding code, other than cosmetic.

Let's take a look at the `_type` node in more detail.

Listing 3.4: Example `_type` node

```
@7  integer_type  name: @10  size: @11  algn: 32
                        prec: 32  sign: signed  min: @12
                        max: @13
```

In this example, we are looking at a node describing an integer type. Every integer type available in C has a link to a node like this (`int`, `long`, `short` but also `char`). We can determine the specific type from the `name` property. In order to do so, we have to go through a `type_decl` node, which also lets us determine if the relevant type was declared using the `typedef` directive (through the presence of `unql` property). Since the Lua C API handles character types differently than numbers, the `size` property is also worth checking — if it points to a constant equal to one byte, then we know that we are working with a `char`. Other `_type` nodes that are commonly present include `real_type` for floating point numbers (`float` or `double`), `pointer_type`, for pointers (including function pointers) and `record_type` for user-defined data structures.

3.3 Handling user defined data structures

As already mentioned, if we stumble upon a `record_type` declaration somewhere inside a chunk, then we know that the function uses a `struct` inside it's body. Further down the line, we will be creating an interface for each structure that is used as an argument or return value from the API calls. Because of this, we need to know the contents of every such data type. Assume that we have a declaration representing a pair in our code:

Listing 3.5: Example `struct` declaration

```
typedef struct {
    int a, b;
} pair_t;
```

In this example we declared `pair_t` as a `typedef` to showcase an important property of the language tree. Now, if the function `foo` uses `pair_t` as one of its arguments, we should be able to see the following nodes inside the AST dump:

Listing 3.6: Example `record_type` nodes for the `pair_t` type

```
@22 record_type name: @29 unql: @23 size: @11
      align: 8 tag: struct flds: @16
@23 record_type size: @11 align: 8 tag: struct
      flds: @16
```

The `record_type` that we will see by following the `tree_list` method described previously, will be the first one listed. Because the `unql` property is present, we can deduce that the struct is in fact declared using a `typedef`. Furthermore, we can access the type name (`pair_t`) by following the `name` node chain. We can also see that there is a property called `flds` — it shouldn't be a surprise that it links to nodes related to the struct fields. Like with function arguments however, we encounter the same obstacle:

Listing 3.7: Example `field_decl` node

```
@16 field_decl name: @21 type: @7 scpe: @23
      srcp: api.h:9 size: @11
      align: 8 bpos: @24
```

We have access only to the first of the two fields of the `pair_t` structure. This time though, we can gather each `field_decl` declaration present in the processed chunk, and connect it with its corresponding data type. The provided `field_decl` example is actually taken from the very same dump file as the example `record_type` nodes. We can see that through the `scpe` property, we can link back to the correct `record_type` declaration (notice the matching index number). There's just a little catch — we can see that the `record_type` that we get in this way is not the same that we access through the `tree_list` traversal. This is because we declared `pair_t` as a `typedef` — without it the process is simplified and we can access the type name straight away. In this case however we need to put extra effort to match these two `record_type` declarations together (using the `unql` property value). After determining the ownership of the processed field, we can access its name and type like previously with function arguments.

3.4 Function pointers

Function pointers introduce us to yet another kind of a `type` node: the `pointer_type` node. Its use is not reserved exclusively for them though; as the name suggests it will be present whenever the source code uses a pointer type. By looking underneath the `ptd` field of this node, we get to the main source of information about the pointer, the `function_type` node.

Listing 3.8: Example `pointer_type` and `function_type` nodes

```
@23 pointer_type size: @24 align: 64 ptd : @31
@31 function_type size: @29 align: 8 retn: @7
      prms: @33
```

Function pointers are much simpler to parse than API calls, as everything we need can be taken from the `retn` and `prms` properties. The return type is hidden beneath the `retn` field — its retrieval is analogous to previously described type extraction. The `prms` property takes us straight to the `tree_list` chain, from which we can extract the exact parameter types.

3.5 The output format

From the beginning the idea was to separate the parser from the binding generator, to allow for some modularity when it comes to different compilers. The only way these two components interact with each other is through an intermediate `.json` representation. A potential developer interested in extending the tool for another kind of compiler is free to approach the AST parsing in whatever way they feel most convenient, as long as the resulting output adheres to the following structure. First of, the top-level JSON object should contain two fields: `Functions` and `Structs`. The value of the first one should be a list of objects containing information about desired API calls. Each object should respect the following convention:

Listing 3.9: Example API `.json` function description

```
1  {
2    "Name": "foo",
3    "Returns": {
4      "Kind": "integer_type",
5      "Typedef": false,
6      "Typename": "int"
7    },
8    "Arguments": [
9      {
10       "Name": "var1",
11       "Kind": "record_type",
```

```
12         "Typedef": true,
13         "Typename": "pair_t"
14     }
15 ]
16 }
```

Each type should be described by its name, family it belongs to (compliant with the GCC way of naming them, like `integer_type` or `character_type`) and whether or not it's declared through a typedef or not. The last property is needed during the binding generation to satisfy the C syntax rules, mainly when working with record types (depending on the situation, adding or omitting the `struct` keyword will result in compilation errors).

The `Structs` field of the top-level object should contain information about each data structure which we want to expose to Lua scripts. We will create a special set of utility functions for each of them — a default constructor and a setter and getter pairs for each field. This means that we need to include the name of the struct and type of all its components.

Listing 3.10: Example `Structs` array entry

```
1  {
2      "Typename": "container",
3      "Typedef": true,
4      "Fields": [
5          {
6              "Name": "c",
7              "Kind": "character_type",
8              "Typedef": false,
9              "Typename": "char"
10         }
11     ]
12 }
```

The structure is analogous to the previously described one with one key difference — this time we need to make sure that we use actual field names instead of auto-generated ones.

Chapter 4

Generating the binding code

4.1 Fetching the arguments

4.1.1 Simple types

4.1.2 C structs and unions

4.1.3 Callbacks and closures

4.2 Error handling

4.3 Exposing the interface

Chapter 5

Example usage

Chapter 6

Conclusion

6.1 Possible further development