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

24 sierpnia 2025

Abstract

Lua is a light-weight, fast and portable scripting language. Because of it, many applications choose to embed its 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

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 [2] 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. Generating the wrappers is not the only thing needed though, as the process of mapping certain types to and from Lua requires additional care — for example, some calls may use user-defined data structures. In that case, we will need to figure out a way to expose them inside the Lua environment. Fortunately, everything needed can be found within the *Abstract Syntax Tree* generated by the compiler.

The binding, when written manually, requires a rather huge amount of repetetive code. This makes the toolchain especially attractive, as its job is to free the programmer of this tedious process.

The project consists of two main parts: the AST parser and the binding generator. Both are written in 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 auxillary 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 that can even use user-defined C data structures.

Chapter 2

Lua C API overview

The 4th edition of the *Programming in Lua* book says:

> Lua is an *embedded language.* This means that Lua is not a stand-alone application, but a library that we can link with other applications to incorporate Lua facilities into them.
>
> [...] This ability to be used as a library to extend an application is what makes Lua an *embeddable* language. At the same time, a program that uses Lua can register new functions in the Lua environment; such functions are implemented in C (or another language), so that they can add facilities that cannot be written directly in Lua. This is what makes Lua an *extensible* language. [1]

Both approaches utilize the so-called *C API* [3] to communicate with Lua. To understand how we can use this API to achieve our goal, we first need to take a look at its capabilities.

2.1 Core concepts

At its core, the C API is just a collection of functions, constants and types that allow C programs to interact with the Lua interpreter. This includes, among other things, running pieces of Lua code and accessing Lua global variables. The main component that allows this type of communication is called the virtual *stack*. From Lua perspective, the stack behaves as we would expect a stack to work — that is the elements are added and removed only from the top of it. From the C perspective however, the 'stack' is more than a traditional LIFO data structure, as random access is possible. Almost every API call operates on values present on the stack, and all data is exchanged through it (from C to Lua and the other way around). Each slot on the stack can hold any Lua value. When we want to get some value from Lua, we can ask for it and receive it on the stack. In a similiar manner, when we want to pass something to Lua, we first have to push it onto the stack. There is a collection of calls doing the pushing and popping operations, defined for different C types. The responsibility of using the correct ones falls on the C programmer. The API also provides a way of manipulating the stack in more *unusual* ways, like replacing or inserting elements in arbitrary places and even element rotation similiar to the one performed bit-wise by assembly instructions like `ROR` or `ROL`. It is also worth noting that the stack is *not* a global structure — each separate interaction (that is for example, a foreign function call) gets its own, local stack.

The stack approach prevents the combinatorial explosion of the API calls (one for each type combination), and also removes any concern about garbage collection — the Lua runtime has exact knowledge about the use of variables stored on the stack and so it will not accidentaly collect them. It also lets us deal with the mismatch between the type systems (dynamic in Lua and static in C). The only price to pay is the need to manually check the correctness of every operation, and having to deal with non-descriptive memory errors when something goes wrong, with the latter being especially common without exerting special care.

2.2 Calling C functions from Lua scripts

For Lua script to be able to call a function written in C, the function must follow a very simple protocol. Let's look at an example provided in *Programming in Lua*[1]:

Listing 2.1: Example C function to be called from Lua

```
static int l_sin (lua_State *L) {
  double d = lua_tonumber(L, 1); /* get argument */
  lua_pushnumber(L, sin(d)); /* push result */
  return 1; /* number of results */
}
```

Every eligible function has to have the same prototype — it has to return an int and accept an argument of type lua_State∗. The single argument is in fact the stack described previously. As we can see, the body of the function consists of:

1. Fetching the passed arguments. This step is more complicated for more complex types, but here we only need a single real number *d*. Each argument, when passed from Lua, gets its unique index on the stack (starting from one and counting up). When fetching the argument we need to specify the correct index. In the provided example, type checking and error handling is omitted.

2. Pushing the result back on the stack. Since we are implementing a simple sine function here, we simply apply it to the argument and push it on top of the stack.

3. Returning the number of 'returned' values. This way the Lua runtime will know how many values it needs to take from the top of the stack (here — only one). After that, the rest of the stack is cleared and the script execution is resumed.

Before we can do anything with this function though, we need to register it. In order to do that, we can mimic the behavior of Lua modules and create our own from within the C code. In order to do that, we have to declare an array describing the module somewhere in our code.

Listing 2.2: Module declaration

```
static const struct luaL_Reg mylib [] = {
  {"sin", l_sin},
  {NULL, NULL} /* sentinel */
};
```

We now have two ways of continuing. We can either link our module dynamically, that is by compiling it into a shared library (`.so`, `.dll`) and including it in a Lua script using the keyword `require`, or register the function as a global Lua variable using a combination of `lua_pushfunction` and `lua_setglobal` API calls. The second approach assumes that we will run the Lua interpreter from within the C program that registers the function. The first approach requires putting an additional function into our source code:

Listing 2.3: The `luaopen_mylib` function

```
int luaopen_mylib (lua_State *L) {
  luaL_newlib(L, mylib);
  return 1;
}
```

This function gets automatically called after including the module with `require`. No matter the approach, we can then call our function from within a Lua script by calling `mylib.sin`.

2.3 Some obstacles and limitations

Even though the API makes exchanging data pretty straightforward, there are still some considerations we need to take into account. First of all, we already mentioned how the type system differs between Lua and C. More specifically, let's look at how we can implement some more complex data structures in both worlds.

In C, we would typically create a `struct` or an `union` to represent a data structure. Then, we could introduce a series of functions doing operations on instances of such data types. In Lua however, we do not have a strict equivalent of C `structs`.

The main way of achieving something similiar is done through the use of Lua's associative tables. This is not what we want though — we want to use actual instances of our API's `structs`, not their cheap knockoffs. The solution is creating default constructors for each introduced data type, along with some utility functions (*setters* and *getters*). This is done through a special API data type called *userdata* and is explained in more detail later. This way we can create `structs` and manipulate them from within Lua.

Another problem is related to passing callbacks from Lua to C. In Lua, functions are *first class citizens* and we can pass them as arguments to other functions, including those introduced by our binding code. For C functions, that is not the case, even though we have something very similiar — function pointers. Our problem is figuring out a way to pass a C function pointer to an API function, that when dereferenced and called executes the function passed on the stack from within a Lua script. This is essentialy a problem of converting a 'reference' to a Lua function to a C function pointer, which turns out is not trivial and is a source of inconvenience and several complications.

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 it's *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 (also, their *order* within the struct),
- For function pointers — essentialy 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 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 funtion 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
                    algn: 8   tag: struct   flds: @16
@23   record_type   size: @11   algn: 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
                  algn: 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 trough 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   algn: 64   ptd  : @31
@31   function_type    size: @29   algn:  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 four fields: `functions`, `records`, `pointers` and `enums`. 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 function `int sum(pair_t p)` description

```
{
  "name": "sum",
  "returns": {
    "kind": "integer_type",
    "typename": "int"
  },
  "arguments": [
    {
      "kind": "record_type",
      "typename": "pair_t"
      }
    ]
}
```

Each type should be described by its name and the family it belongs to (complacent with the GCC way of naming them, like `integer_type` or `character_type`). The `typename` property should include a relevant prefix (like `struct`, `union` or `enum`) when needed, to satisfy the C syntax rules (when the corresponding data type is declared without the use of the `typedef` directive, omitting the prefix will result in compilation errors). We showed previously, how we can determine whether or not such prefix should be present.

The `records` 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/union and type of all its components.

Listing 3.10: Example `records` array entry

```
{
  "kind": "record_type",
  "typename": "struct container",
  "fields": [
    {
      "name": "character",
      "kind": "character_type",
      "typename": "char"
    }
  ]
}
```

Listing 3.11: Corresponding `struct container` definition

```
struct container {
  char character;
};
```

The structure is similiar to the previously described one with one key difference — this time we need to make sure that we use actual field names under the `name` property of each field. We can also see here, that since `container` is declared without the use of `typedef`, the `struct` prefix is present in the type name.

The `enums` field should contain information about each enumeral type that we wish to use inside the API calls. Once again, its structure is pretty much analogous, with the difference being that this time a field is described by its name and value.

Listing 3.12: Example `enum` declaration

```
typedef enum {
  RED=42, GREEN, BLUE
} RGB;
```

Listing 3.13: Fragment of the corresponding `enum` description in the resulting `.json`

```
{
  "typename": "RGB",
  "fields": [
    {
      "name": "RED",
      "value": 42
    },
    [...]
  ]
}
```

Lastly, the `pointers` field should contain information about each pointer type found throughout the AST dump. Theoretically, we could attach this information directly to the argument/field descriptions found throughout the `.json` file (consequently, we could do the same with `enums` and `records`), but this would introduce unnecessary repetition to the result. More importantly, this could cause an infinite recursive relation inside the result — just imagine how would a data type like this be represented:

Listing 3.14: `recursion_t` data type causing problems for the parser

```
typedef struct Recursion {
  struct Recursion *twin;
} recursion_t;
```

That's why, a single pointer description contains information about the type only a single level beneath the described pointer (which could very well be another pointer, described in a separate entry). An example of the produced output should be enough of a clarification for the structure of each entry:

Listing 3.15: Several example entries inside the `pointers` array

```
{
  "typename": "int (*)(int)",
  "underlying": {
    "kind": "function_type",
    "typename": null,
    "returns": {
      "kind": "integer_type",
      "typename": "int"
      },
      "arguments": [
        {
          "kind": "integer_type",
          "typename": "int"
        }
      ]
    }
  },
  {
    "typename": "int*",
    "underlying": {
      "kind": "integer_type",
      "typename": "int"
    }
  }
```

Chapter 4

Generating the binding code

While going over the possibilities of the Lua C API (chapter 2.2) we saw an example `sin` function that could be called from a `.lua` script. All generated wrappers will follow the same three-step formula as that example function — prepare the arguments, pass them to the API call and push the result onto the stack. The regularity of this structure is actually what makes the whole process so atractive in terms of code generation. Our main focus will lay in figuring out how to actually prepare the arguments of various types.

Besides wrappers, we will also need to implement some core interface functions, like initializing the Lua enviroment, running scripts and cleaning memory after we are done. Parts of the initialization process will also need to utilize generated helper functions as well as module-defining registers.

4.1 Some technicalities

Since our ultimate goal is generating a `.c` source file, it would make sense to talk about how we can do that first. Obviously, we could generate our code using pure Python using simple, core mechanism like string formatting, but that seems painfully tedious and unmaintainable. Instead, we will use a templating engine called `Jinja` [6]. With it, we can create a set of templates resembling actual code in C, with some elements being dependent on data we pass from the main application. We can then combine those templates into the final product. The ability to utilize simple control structures and macros inside the templates make them a very powerful and useful tool in the process of code generation. They also make the whole ordeal much simpler, as we are basically coding in C — moreover, we have precise control over the formatting to make the result look fine without the need to use external formatters, or worry about implementing custom logic. This approach makes everything much more readable and keeps it modular, as a single component that we use many times throughout the binding can be moved into its own template.

Listing 4.1: Example Jinja C template.

```
{%- import "macros.c.j2" as M with context -%}
{% set T = record["typename"] %}
int cbind_{{T}}_new(lua_State *L) {
  {{T}} *result =
    ({{T}}*)lua_newuserdata(L, sizeof({{T}}));
  {%- if record["kind"] == "record_type" %}
  {%- for field in record["fields"] %}
  {{M.fetch("L",loop.index,"arg"~loop.index,field)}}
  result->{{field["name"]}} =
    {{M.pass("arg"~loop.index, field)}};
  {%- endfor %}
  {%- endif %}
  luaL_getmetatable(L, "{{T}}");
  lua_setmetatable(L, -2);
  return 1;
}
```

4.2 Simple types

Handling simple C types is just as the name suggests — simple. Types like `int`, `float`, `char` and `const char*`, but also their extensions like `long` and `double` translate pretty much directly into our binding code. For all kind of integers, we can utilize the `lua_tointeger` call, to fetch the "integer compatible" value from the virtual stack — this means that the call will work for real numbers as well as strings convertible to numbers. For real numbers we have `lua_tonumber` call, which works in the same way. For strings, we can use `lua_tostring` function. As there's no distinction between a string (`char*`) and a single character (`char`) in Lua, we need to use the same call for characters as well, with the addition of taking the first character from the fetched value. We need to also keep in mind that because of garbage collection, the returned pointers may not be valid after removing the value from the stack.

Additionaly, as `enums` are typically represented by 32-bit integers in C, we can treat them as such in the fetching process. This means that other than exposing them to the Lua environment, we needn't put any more special care in their handling.

4.3 C structs and unions

Records are the first instance of types that need special care in the binding code. We mentioned previously that, we will be creating a set of utility methods and a default constructor for each of them. This is because they will be represented by a special Lua type called `userdata`. According to *Programming in Lua*:

> A userdata offers a raw memory area, with no predefined operations in Lua, which we can use to store anything. [1]

As such, the default constructor will consist of creating an userdata and populating it with the record field values. That way, everytime a record is used in an API call, we will expect an `userdata` value on the stack (we can get it with `lua_touserdata`).

We would also want to be able to access specific fields of the created record. To achieve that functionality, each record will be equipped with a set of access methods for each field it contains. For convienience reasons, we will utilize a mechanism called the `__index` methamethod, fully described in the *Object-Oriented Access* section of chapter 31 of *Programming in Lua*. That way, we enable the record:get_x() syntax — instead of module.get_x(record). Lastly, we should also have a way of distinguishing between instances of different types. This is important for type checking — after all, so far all records are represented using the same `userdata` type. This is solved using *metatables* — from *Programming in Lua* again:

> Metatables allow us to change the behavior of an [userdata] For instance, using metatables, we can define how Lua computes the expression a+b, where a and b are [userdata]. Whenever Lua tries to add two [userdata], it checks whether either of them has a metatable and whether that metatable has an `__add` field. If Lua finds this field, it calls the corresponding value (the so-called metamethod, which should be a function) to compute the sum. [1]
>
> The usual method to distinguish one type of userdata from another is to create a unique metatable for that type. Every time we create a userdata, we mark it with the corresponding metatable; every time we get a userdata, we check whether it has the right metatable. Because Lua code cannot change the metatable of a userdata, it cannot deceive these checks. [1]

We can mark an userdata with a metatable during the process of its creation, that is inside the default constructor. The process of creating such metatable as well as enabling the object-oriented access mentioned before will be handled by a helper function for the initialization function, which is part of the main interface.

4.4 Function pointers

This is by far the most interesting part in the whole process. First, let's break it down into smaller parts. On the stack, we are presented with a *function* value. The

API call expects a function pointer — a location in memory of something that we can *call*. Glossing over the Lua C API however, we see that a simple conversion (like with simple types) is not possible. This shouldn't be surprising, as functions in C are something inherently different than values (unlike in Lua, where functions are first-class citizens). We need to point to *something* though — something that when called will behave exactly like the supplied argument.

To actually call the function passed from a Lua script, we can utilize the `lua_pcall` call, from the Lua C API. The process is simple — with the desired function on top of the stack, we need to push its arguments (in the correct order) and call `lua_pcall`, passing the number of arguments supplied and the number of return values that we expect. Since functions in C can return only a single value (or not return anything at all), the second value will always be equal to one or zero. The call itself will pop the function along with its arguments and push the returned values on top of the stack. We can then retrieve the result just like we would with any other type discussed previously.

We are now presented with a simple idea — what if we created a *template* function, with the desired signature that does the exact thing we just described. After all, we have knowledge about every function pointer kind used throughout the target API — for each of them we could create a separate template. We would then have a function that we can point to, and consequently we could pass a pointer to it to the API call each time it's required. There's one problem though — how would the actual function know which Lua function we want to call? Since we don't know how many different functions will be used by the `.lua` scripts beforehand, we can't create a dedicated template for each of them. This means however, that every function with the same signature will have to utilize the same template function.

One idea would be to create some kind of a global reference, that the templates would use to fetch the desired function. A single reference would not be enough, as we have no control over the order or time that these functions will actually be called by the API (the *event scheduler* concept presents that clearly). Multiple references don't solve anything as well — we still wouldn't know which one to choose.

A solution presents itself in the form of *closures*. A closure [8] is a function along with the environment required to run it. If we could create such closure during runtime, using the templates from the original idea and passing a reference to the Lua function inside their environment, we could achieve what we want.

Listing 4.2: Example of a generated template function for an `int (*)(int)` callback

```
void i_i(void *data, va_alist args) {
  /* accessing closure data */
  closure_t *c = (closure_t *)data;
  lua_State *L = (lua_State *)c->L;
  lua_rawgeti(L, LUA_REGISTRYINDEX, c->key);
```

```
    /* argument fetching */
    va_start_int(args);
    int arg1 = va_arg_int(args);
    lua_pushinteger(L, arg1);

    lua_pcall(L, 1, 1, 0);

    /* returing a value */
    int result = lua_tointeger(L, -1);
    va_return_int(args, result);
}
```

4.4.1 Closures with *GNU libffcall*

Since closures in the form we just described are not present in standard C, we have to resort to another form of achieving them. The *GNU libffcall* [4] library does just that. We can create a closure by calling the `alloc_callback` function and supplying it with an address of a template function and a pointer to the data that we want inside its environment. This call creates a small piece of architecture-dependent machine code using `mmap` — a trampoline — that behaves as a regular function pointer. Its job is to provide the handler (the function we passed to `alloc_callback`) with the supplied arguments (at the time of calling it), as well as the previously passed closure data. After being done with the created closure, we can clear the memory it occupies by calling `free_callback` on the pointer returned from the initial call.

To prevent memory leaks, each time a closure is created the binding code stores every pointer relevant to it, so that they can later be properly freed. The storage is implemented using an AVL tree for maximum efficiency. Since functions called from Lua can return many values, each API call that uses function pointers returns the registry keys associated with closures created during its execution. An additional function exposed to the Lua environment called `delete_callback` can be called to prematurely free the memory related to its single argument — the aformentioned registry key. The whole tree is destroyed upon calling the main interface function `cbind_close`.

The library also provides an interface for creating functions with variable argument/return count, but that is more of an obstacle in this scenario. It doesn't help that the documentation is really brief and does not go into any details either, but looking into the source code and some experimentation proved that it is suitable for the job. It was also already indirectly mentioned, but its worth noting explicitly that for it to work, it needs to be able to allocate pages that are both *writable* and *executable*.

4.5 Regular pointers

When dealing with pointers to memory allocated on the C side, Lua offers yet another special type — *light userdata*.

> A light userdata is a value that represents a C pointer, that is, a `void*` value. A light userdata is a value, not an object; we do not create them (in the same way that we do not create numbers). To put a light userdata onto the stack, we call `lua_pushlightuserdata`:
>
> ```
> void lua_pushlightuserdata (lua_State *L, void *p);
> ```
>
> Despite their common name, light userdata and full userdata are quite different things. Light userdata are not buffers, but bare pointers. They have no metatables. Like numbers, light userdata are not managed by the garbage collector. [1]

Functions returning a pointer should then encapsulate them inside a light userdata container. The rule to follow here pretty much comes down to avoiding mixing responsibility — resources managed by C should stay managed by it, and vice versa. Using light userdata we can pass around C-managed pointers to other API calls.

If we want to create a pointer on the Lua side, we go back to full userdata. Since we don't want to allocate extra resources on the C side, all we do is wrap the requested value in a userdata container, and return it back to Lua. This way, it stays managed by the Lua environment. An extra utility function exposed by the binding code serves that purpose by additionally attaching a metatable tag to the created pointer. That way, when we need to dereference the value, we can cast the pointer to a proper data type.

Listing 4.3: Creating an `int` pointer in a `.lua` script

```
local value = 42
local ptr = API.wrap(value, API.ctype.INT)
-- assert(API.unwrap(ptr) == 42)
```

4.6 Error handling

When dealing with the Lua C API, the responsibility of type checking falls on the C programmer. Failure to do so could lead to serious runtime errors. As such we would like to have a mechanism for ensuring that arguments passed from Lua won't cause any problems on the C side.

A standard way of informing Lua that something went wrong, is invoking the `luaL_error` function with the appropriate error message. This raises a runtime error within a Lua script that causes it. Even though it never returns, it's customary to invoke it inside a `return` statement. We are also equipped with a set of `lua_check*` functions, which we can supply with an appropriate stack index, to check whether or not the value stored there satisfies our needs. This is straightfoward for simple types; for records we can use a function called `luaL_checkudata` which also accepts a metatable name.

Unfortunately though, type checking functions is very limited. Even though we can call `lua_isfunction`, all it does is assure us that we did in fact receive a function on the stack. We have no way of checking the number of arguments it receives, their types and the return value type. Fortunately, when talking about callbacks in previous section, we mentioned how the actual function is called through `lua_pcall`. This is good, because the actual call is done in a *protected mode*. The function can fail for whichever reason, but it won't crash the whole program. A single drawback is that we seemingly can't propagate the failure reason in any way, even though we have access to it. Other than that, the fact that something went wrong can be reflected in the return value, which is the way most code in C does it anyway.

4.7 Exposing the interface

The main interface exposed by the binding code consists of three calls: `cbind_init`, `cbind_execute` and `cbind_close`. Before any interaction with Lua scripts can happen, the main C application needs to call `cbind_init`. It is responsible for opening required libraries and initializing the environment, populating it with created wrappers and utility functions. Calling it returns a handler which can then be used in the other two calls. Its type is purposefully `void*`, so that the potential user doesn't need to worry about the innerworking of the binding code.

Listing 4.4: Interface singatures

```
void *cbind_init(const char *module);
int cbind_execute(void *state, const char *file);
void cbind_close(void *state);
```

The `cbind_execute` call is used to execute `.lua` scripts in the created environment. It returns a status indicating whether or not everything went smoothly.

Lastly, the `cbind_close` call is used to finalize interaction with the Lua interpreter. After calling it, the environment is no longer usable, which consequently means that every registered callback gets destroyed. It makes sure that the extra allocated memory is freed properly, and that the environment gets gracefully closed.

Listing 4.5: Example interaction with the interface

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include "binding.h"

int main(int argc, char *argv[]) {
  const char *file = argv[1];
  void *state = cbind_init("API");
  int status = cbind_execute(state, file);
  cbind_close(state);
  if (status != 0)
    exit(EXIT_FAILURE);
  exit(EXIT_SUCCESS);
}
```

Chapter 5

The project repository

Lastly, we will go over the project structure and the contents of the repository its contained in.

The project repository consists of following directories/files: The `cbind` directory contains the source code for the parser and the binding generator, with the `cbind/templates` subdirectory containing all C templates used the generation process. Next up, the `doc` directory contains the `thesis.tex` file used in the creation of this PDF, with the PDF itself in the same directory. The `examples` directory contains an example application used to demonstrate the abilities of the toolchain, and is described later in this chapter. After that, there is the `tests` directory, consisting of the `tests/include`, `tests/scripts` and `tests/src` subdirectories, as well as a `Makefile` and a validation script `validate.sh`. The listed directories contain the header files for the tested API, its implementation (inside the `tests/src` directory) and several test cases in the form of `.lua` scripts (the `tests/scripts` directory). In the `tests/src` directory, there is also a main application code, that is used to run the aforementioned scripts. Lastly, in the project's root directory, there is a `README.md` file going over the core workings of the project, as well as a `Dockerfile` used in automatic testing.

The approach to testing is closest to the *Test Driven Development* philosophy, as the test cases were created before implementing each major feature of the binding generator, and were the deciding factor in determining whether or not the feature works. As the repository is stored on `GitHub`, the *GitHub Actions* faculty was integrated into the project. This means, that on every push, the tests are automatically ran inside a docker container to confirm that everything works as expected in a fresh environment. The details for the way that is achieved can be found inside the `.github` directory and the mentioned `Dockerfile`.

5.1 Example usage

To see the toolchain abilities in action, we can go through an extremely simple example program. The whole application is supposed to look like a very early prototype of a video game, written using the SDL2 [5] library. In it, the player (represented by a green rectangle) can navigate a 5×5 grid and execute their signature attack (represented by other rectangles flashing). The attack pattern and its color are dictated by two variables: color_t *attack_color and **int** (*attack_pattern)(**int**, **int**). The program is written in a way so that these two variables are to be set by an external `.lua` script. For this purpose, several API calls were created: one for setting the color of the attack, and one for setting the attack pattern itself. Additional calls supply the scripts with the current player position and the size of the grid (so it can be changed freely without breaking any logic depending on it).

Listing 5.1: `color_t type definition`

```
typedef struct {
  short red, green, blue;
  short alpha;
} color_t;
```

The type of the `attack_pattern` variable suggests that the pattern is represented by a characteristic function of a set of coordinates. When prompted with the x and y grid coordinates, it tells the main program whether or not the corresponding square is being attacked by the player. The function can depend on the player's position, but it can also ignore it completely when going through its logic.

Listing 5.2: Example pattern dictating function

```
function pat(x, y) -- cross shape around the player
  local posx, posy = API.get_posx(), API.get_posy()
  if math.abs(x-posx) + math.abs(y-posy) == 1 then
    return 1
  else return 0 end
end
```

The interaction with the program is done by pressing certain keys on the keyboard — arrow keys are for movement, the space bar launches the currently set attack and the *r* key is used to dynamically reload the settings. The *r* key is especially interesting, as it allows us to change the attack patterns without closing the game window, simply by changing the code contained in the `script.lua` file. After executing the parser and the binding generator on the existing source files, we end up with the binding code consisting of roughly *600 lines*, allowing us to freely call the aforementioned API calls, and play around with the attack settings.

5.1.1 Running the example

To run the example application yourself you should follow this simple guide. First of, we will go through the process of installing the dependencies. Make sure that your system has the standard compilation tools available to use — mainly those contained in packages like `base-devel`. A Python installation should also be present.

For Python, the required libraries are listed under the `requirements.txt` file — to install them using `pip`, the `pip install -r requirements.txt` command is used inside the root directory. For convenience reasons, this command should be ran inside the Python's *virtual environment*. To create such environment, the `python -m venv /path/to/new/virtual/environment` command should be ran, just as the official documentation [7] says. Once created, run the `source <venv>/bin/activate` to activate it. Installing Python packages inside the virtual environment prevents clashes with already installed packages, and polluting the base environment.

Next up, we will need a Lua installation. Since the building process is extremely simple and is actually the recommended approach, one should follow the building instructions listed on the official site [2]. After installing Lua, the `-llua` linker flag becomes available — we will need it in the compilation process.

For C dependencies we will need the `libffcall` [4] library and the `SDL2` [5] library installed. Both of those should be available for installation through the systems package manager — commands like `pacman -S sdl2 ffcall`. Once again, we will need the `-lffcall` and `-lSDL2` linker flags.

After installing the listed dependencies, navigate to the `examples` subdirectory, and run the `make compile` command. This will create the `examples/main` executable which can then be freely played around with. The process of running tests is similiar in a way that it need the same set of dependencies installed (besides the `SDL2` library, which is only used in the example program). To run them, navigate to the `tests` directory and execute the `validate.sh` script. This will run each written test and display the number of successful ones in the terminal.

Chapter 6

Conclusion

During my work on the project, I have successfully implemented a toolchain consisting of two independent components: a GCC abstract syntax tree parser and a binding generator operating on its result. The toolchain was tested against variety of scenarios and corner cases. An example application was also provided to show, that it can be proven useful in certain scenarios

The first toolchain component (the parser) required the understanding of a complex, yet comprehensible compiler representation of the target library code. It also required choosing the one most suitable, as there were a lot of them available. Many experiments driven by trial and error helped to make the chosen representation clearer and ultimately useful in the process of generating the binding code. The intermediate result produced by the parser was also carefully designed, so it could cooperate with the binding generator as best as it could.

The second component required knowledge about the abilities of the Lua C API, and some creativity when dealing with the problem of mapping C types to those available in Lua. This part was also riddled with experimentation, which had the goal of measuring what's actually possible, and what's not. These experiments lead to an extensive usage of the offered API, and some interesting solutions — like using closures.

6.1 Possible further development

As mentioned in the beginning, the parser was developed as an interchangeable component from the start. A nice addition to the toolchain would be to create additional ones for other compilers, mainly `clang`, which offers an extremely convenient option of dumping the AST in a complete, readable way. The ability to switch between compilers would be beneficial for the user, who would not be forced to use one over another. The additional parser could also help deal with some problems arising with the use of the GCC's AST — like ones related to structs. It turns out that the compiler ommits certain struct fields in the AST, if they are not used in any defined functions. The severity of this problem is not that big however, as a simple *dummy* function referencing each field fixes it.

Another improvement that could be implemented relates to the usage of multiple threads. As of now, the binding may not be thread-safe, specifically when working with callbacks. This is because the callback storage described in subsection about closures (chapter 4.4.1) does not account for multi-threading. Additionally, some logging functionality could be introduced to the binding code, to optionally report problems arising from incorrectly typed callbacks.

Lastly, everything related to regular pointer values could possibly be improved over the need to manually specify the type of the pointer. As of now, this can be seen as just an inconvenience for the user, but can also possibly cause some problems when used incorrectly.

Bibliography

[1] Roberto Ierusalimschy, *Programming in Lua*, Feisty Duck Ltd, 4th edition, 2016.

[2] PUC-Rio, *The Programming Language Lua*, Web: `https://www.lua.org/about.html`.

[3] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, *Lua 5.4 Reference Manual*, Web: `https://www.lua.org/manual/5.4/manual.html`.

[4] Free Software Foundation, *libffcall — GNU Project — Free Software Foundation*, Web: `https://www.gnu.org/software/libffcall/`.

[5] *Simple DirectMedia Layer*, Web: `https://www.libsdl.org/`.

[6] Pallets, *Jinja — Jinja Documentation (3.1.x)*, Web: `https://jinja.palletsprojects.com/en/stable/`

[7] Python Software Foundation, *venv — Creation of virtual environments*, Web: `https://docs.python.org/3/library/venv.html`.

[8] Wikimedia Foundation, *Closure (computer programming)*, Web: `https://en.wikipedia.org/wiki/Closure_(computer_programming)`.