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

11 kwietnia 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 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 it's 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 it's 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 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 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

3.1 Example overview based on the *gcc* compiler

Chapter 4

Parsing the AST

4.1 Goal and the output format

4.2 Function arguments and their types

4.3 Handling user defined data structures

4.4 Function pointers

Chapter 5

Generating the binding code

5.1 Fetching the arguments

5.1.1 Simple types

5.1.2 C structs and unions

5.1.3 Callbacks and closures

5.2 Error handling

5.3 Exposing the interface

Chapter 6

Example usage

Chapter 7

Conclusion

7.1 Possible further development