Felipe V. Côrtes

Extrator de Tipos para Lua

Projeto Final de Programação

Especificação apresentada como requisito parcial para obtenção de grau da disciplina Projeto final de Programação em Informática, do Departamento de Informática da PUC-Rio .

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
February 2023

Abstract

Inspecting dynamically typed code is hard due to the lack of type information provided. Inspired by this challenge, we built a tool capable of inspecting Lua programs and extracting types from the functions in it. Sustained by the reflection capabilities of the language, it's possible to extract parameter and return values from each function execution and generate a useful report for code documentation and inspection. This document presents the software design and implementation, as well as results obtained by some Lua benchmark programs.

Keywords

Resumo

Inspecionar código dinamicamente tipado é uma tarefa difícil devido a falta de informação sobre tipos. Inspirado por esse desafio, construímos uma ferramenta capaz de inspecionar programas feitos em Lua, extraindo os tipos das funções nele presentes. Sustentado pelas capacidades introspectivas da linguagem, é possível extrair os valores dos parâmetros e retornos de cada execução de função, gerando um relatório útil para documentação e inspeção. Esse documento apresenta a especificação e implementação do software, assim como os resultados obtidos em alguns programas Lua de referência.

Palavras-chave

Table of contents

List of codes

1
Introduction

There are several reasons that motivate the adoption of statically typed languages. Maintaining large systems built with dynamic types can become a nightmare due to the lack of type information (TAKIKAWA et al.,). Typed languages has also generally better performance because compile-time type information helps generating optimized machine code. However, programmers are frequently left empty-handed when inspecting dynamically typed code while having to re-write systems to a statically typed languaged if gradually typed languages are not an option.

Inspired by the challenge of inspecting dynamically typed code, we built a type extractor for the Lua programming language. By inspecting a program's execution during runtime, it can generate a detailed report with the types being transfered between functions. We believe that the type extractor can help programmers understand the types in a program's execution, contributing for code documentation, debugging and optimization.

The document is structured as follows. In Chapter 2 we present previous work related to type systems in Lua. In Chapter 3 we describe the software goal. Chapter **??** explain the software modules and how they interact. In Chapter **??** it's shown the software key functions, the modules relationship and basic utilization. In Chapter **??** we present and discuss some results obtained by the type extractor with some Lua benchmarks. Finally on Chapter **??** we present our conclusion and future work.

2
Previous Work

There has been some notable works about Lua type system that we must cite. Typed Lua (MAIDL; MASCARENHAS; IERUSALIMSCHY, 2014) has already defined an optional type system for the language. More than enriching documentation, this extension ensures static type safety while preserving Lua idioms. Typed Lua encodes the main data structure mechanism from Lua into arrays, records, tuples and maps. It uses a bracket syntax to denote table types:

Code 1: Insert Typed Lua

```
1 local interface Element
2     info:number
3     next:Element?
4 end
5
6 local function insert (e:Element?,v:number):Element
7     return { info = v, next = e }
8 end
```

The type system is designed to be lightweight and type-safe and extends for typing object, classes and modules by adding type annotations. In Code 1 example, a simple algorithm for inserting numbers in a list is shown using type annotations. The Element interface is defined recursively and referenced twice on the function's header, indicating it's return type. The $?$ symbol means that e is optional and can assume empty values. Although Typed Lua's type system share some parts with other optional type systems for dynamically typed languages, it's design demanded uncommon features due to Lua's characteristics.

Lua Type System has also been explored for scripting optimization with Pallene (GUALANDI; IERUSALIMSCHY,). The language design is inspired by optional type systems and it's semantical and syntatical similarity with Lua enables integrating seamlessly with Lua's dynamic code.

Code 2: Pallene Array Sum

```
1 function sum ( xs : { float }): float
2     local s : float = 0 .0
3     for i = 1 , # xs do
4         s = s + xs [ i ]
5     end
6     return s
```

```
7 end
```

As opposed to Typed Lua, Pallene is designed for efficiency. It performs runtime checks to ensure type safety with a particular flexibility. Similarly, Pallene uses type annotations. As shown in Code 2, the function *sum* receives an array of float and returns a single float. Pallene has a built-in interoperability with Lua by sharing its runtime and data-structures. These features allow converting Lua code to Pallene code more easly.

3
Project Scope

Type extraction for dymanic code is still a low covered subject. Our goal is to build a complementary tool, collecting type information from an user's program and reporting this data for documentation, inspection and code migration. The reflection properties of Lua allow us to register hook functions and extract the values contained in the program's execution by accessing the values during runtime. As an output the program generates a list of function types containing all gathered information. Giving tools for understanding the types relations inside a program helps programmers to debug and optimize dynamically typed code.

Lua values can assume several types, speacially tables, which is the main data-structure mechanism of the language, and functions, considered as first class values. This type dynamism makes type inspection a challenging task, so in order to reduce some complexity, we chose to follow a merge strategy for types following the Pallene Language type specification. Pallene conventional type system brings simplicity for table types, restricting them as array types and record types and shows a straightforward function type definition. Serving as an analysis tool, we won't make any type verification or restrain the program's execution. The types that could not be infered will be shown as a dynamic type.

The extractor offers two ways to inspect functions inside a program. A full program inspection is available by passing a lua program as input to the extractor. In this approach, each Lua function will be analysed. An alternative way is to import the extractor as an auxiliar library. By importing the inspection library, the programmer can register specific functions for inspection and select what part of the program they want to analyse.

4
Bibliography

GUALANDI, H. M.; IERUSALIMSCHY, R. Pallene: A companion language for lua. v. 189, p. 102393. ISSN 01676423. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167642320300046>.

MAIDL, A. M.; MASCARENHAS, F.; IERUSALIMSCHY, R. Typed lua: An optional type system for lua. In: **Proceedings of the Workshop on Dynamic Languages and Applications**. New York, NY, USA: Association for Computing Machinery, 2014. (Dyla'14), p. 1–10. ISBN 9781450329163. Disponível em: <https://doi.org/10.1145/2617548.2617553>.

TAKIKAWA, A. et al. Is sound gradual typing dead? In: **Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. Association for Computing Machinery. (POPL '16), p. 456–468. ISBN 978-1-4503-3549-2. Event-place: St. Petersburg, FL, USA. Disponível em: <https://doi.org/10.1145/2837614.2837630>.