Pontifícia Universidade Católica
do Rio de Janeiro

Felipe V. Côrtes

Type Extractor

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
March 2018

Abstract

Cortes,Felipe; Ierusalimschy, Roberto (Advisor). **Type Extractor**. Rio de Janeiro, 2018. 18p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

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

Lua; Programming Languages; Type Systems; Code Inspection.

Resumo

Cortes,Felipe; Ierusalimschy, Roberto. **Extrator de tipos**. Rio de Janeiro, 2018. 18p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

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

Lua; Programming Languages; Type Systems; Code Inspection.

Table of contents

List of figures

List of tables

List of algorithms

List of codes

List of Abreviations

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 enough information to help programmers visualize the types being transfered between functions of their program. The software output can be used as an useful documentation, while also helping programmers migrate code to a statically typed one or even for debugging.

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 4 explain the software modules and how they interact. In Chapter 5 it's shown the software key functions, the modules relationship and basic utilization. In Chapter 6 we present and discuss some results obtained by the type extractor on some Lua benchmarks. Finally on Chapter 7 we present our conclusion and future work.

2
Previous Work

There has been some notable works about Type System with Lua 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

```
local interface Element
    info:number
    next:Element?
end

local function insert (e:Element?,v:number):Element
    return { info = v, next = e }
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

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

```
    return s
end
```

As opposed to Typed Lua, Pallene is designed for efficiency. It performs runtime checks to ensure type safety with a tweak flexibility. But 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 existing dymanic code is still a low covered subject. Our software has the purpose of 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 these values during runtime, the extractor builds a structure containing all type information gathered so far.

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 specification brings simplicity for table types, which are divided between array types and record types.

The type extractor can be used by two approaches, as a full program analysis or as a inspection library.

– Full Analysis: A full program analysis is made by passing a Lua program as input to the extractor. In this approach, each possible function call and return types will be analysed.

– Inspection library: An auxiliar library, capable of registring specific functions for inspection. In this approach, the programmer can select what part of the program they want to analyse.

These usage scenarios enables the extractor to be used as an auxiliary tool for migrating from dynamically to statically typed languages. At the end, a report containing information about parameters and return types of each analysed function is generated and shown to the user. This data serves as a good documentation for functions parameter and return types. Giving tools for understanding the type relations inside a program helps programmers to debug and optimize dynamically typed code.

4
Project Specification

5
Development

6
Results

7
Final Considerations

8
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>.