Pontifícia Universidade Católica
do Rio de Janeiro

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

Cortes, Felipe; Ierusalimschy, Roberto (Advisor). **Extrator de Tipos para Lua**. Rio de Janeiro, 2023. 19p. 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; Language; Type; Inspection.

Resumo

Cortes, Felipe; Ierusalimschy, Roberto. **Extrator de Tipos para Lua**. Rio de Janeiro, 2023. 19p. 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 funções 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; Language; Type; Inspection.

Table of contents

List of figures

List of tables

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 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 with some Lua benchmarks. Finally on Chapter 7 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 a challenging task. This project explores the reflective abilites of Lua to achieve the goal of building a complementary tool to collect type information from an user's program and report this data for documentation, inspection and code migration. The instrospective functions of Lua debug library allow us to inspect names and values of a running program. It also provides a hook mechanism for registering functions to trace the program's execution. As an output the program generates a list describing the function types for parameter and return values.

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 this 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.

Differently from a type checking algorithm, the type extractor won't make any type validation or enforce type constraints. It is designed to analyse the types contained in a program's execution and report this information as a readable report. It offers two ways of program inspection. A full program inspection, when the user passes a lua program as input to the extractor. In this approach, each Lua function called during the execution 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. Designed to be an analysis tool, it provides better understanding of the types relations in a program, helping programmers to debug and optimize dynamically typed code.

4
Project Specification

The objective of the type extractor is to generate a readable report for the user containing the types of parameter and return values of each function in a program's execution. With this objective in mind, we explored the reflection abilities of Lua through the following modules.

Type

The Type module is responsible for categorizing Lua values into a refined type representation. As said before, our type representation is borrowed by the Pallene type system and can represent seven different categories:

$\tau := nil \mid boolean \mid integer \mid float \mid number \mid string$	primitive types
$\mid \{\tau\}$	array type
$\mid \{l_i : \tau_i\}^{i \in 1..n}$	record type
$\mid \tau_i^{i \in 1..n} \to \tau_j^{j \in 1..m}$	function type
$\mid \tau?$	optional type
$\mid \{\}$	empty type
$\mid any$	dynamic type

It also defines a union strategy between two types. If types are equal, then the result of a union is of the same type but if types are incopatible, a dynamic type is generated. A compatibility table is shown in 4.1 to help understanding this union strategy. Table 4.2 shows the union strategy between primitive number types.

Table 4.1: Compatibility table

	nil	boolean	integer	float	number	string	array	record	empty	function	any
nil											
boolean											
integer											
float											
number											
string											
array											
record											
empty											
function											
any											

Table 4.2: Primitive Number Type Union

type1	type2	result
integer	float	float
integer	number	number
float	number	number

For recursive types, optional type and dynamic type, the following definition states the rules for union operation:

$$\{\tau_i\} \cup \{\tau_j\} = \{\tau_i \cup \tau_j\} \qquad \text{array union}$$

$$\{l : \tau_i\} \cup \{k : \tau_j\} = \begin{cases} \{l_i : \tau_i \cup \tau_j\} & \text{if } l = k \\ \{l : \tau_i \cup nil, k : \tau_j \cup nil\} & \text{if } l \neq k \end{cases} \qquad \text{record union}$$

$$\tau_i^{i \in 1..n} \cup \tau_j^{j \in 1..m} = \begin{cases} \tau_i \cup \tau_j & \text{if } i = j \\ \tau_i \cup nil & \text{if } i > m \\ \tau_j \cup nil & \text{if } j > n \end{cases} \qquad \text{function union}$$

$$\tau \cup nil = \tau? \qquad \text{nil union}$$

$$\tau \cup any = any \qquad \text{any union}$$

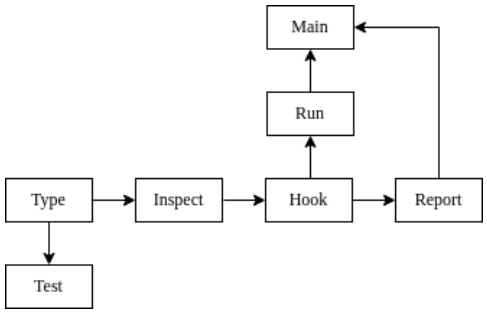

Figure 4.1: Diagram module

Inspect

The Inspect module is responsible for accessing the parameter and return values, analysing each one by our type definition. In order to access return values, the inspection module is dependent on the version 5.4 of Lua, which enables the inspection of transfered values through the debug library.

Hook

A hook function is a piece of code to be executed during specific events. There are several events available to inspect but the extractor is focused on inspecting call and return events. The Hook module is resposible for configuring the inspection function to be executed at these events.

Report

Generating a report consists in transforming the type information collected so far to a human readable output. The Report module is responsible for printing some function informations as well as the function types.

Figure 4.1 shows the relationship between the project modules. Other than the modules already described here, there are modules responsible for configuring and running the extractor. Specially, the Test module validates the definitions described earlier. It asserts that the result of a type creation and type union is the one expected by an equality function between types. A type is equal to another if it has the exact same definition.

5
Development

The type extractor depends heavily on the Lua debug library. Our tool make use of the hook mechanism and introspective functions of the language to inspect names and values inside a program execution. In this section we will describe the implementation of our extractor, emphasizing some key parts.

Basic profiler

The project development is inspired by a rudimentary profiler specified in Chapter 25 of the Programming in Lua book (IERUSALIMSCHY,). This profler gives us insight of how a function can be inspected during runtime and can be easly expanded to explore other introspective capabilities. Code 3 shows the hook function registered to be executed by each event. Inevitably, some functions we do not want to inspect are captured by the this hook, so we ignore these functions as soon as we identify it's not a Lua function. The counterside of this restriction is that C functions defined by the user will not be inspected.

Code 3: Hook function

```
1  function Hook (event)
2      local infos = debug.getinfo(2,"Snfrt")
3      local f = infos.func
4      if (Ignores[f] == true) then
5          return
6      else
7          if(Counters[f] == nil) then
8              if (infos.what ~= "Lua") then
9                  Ignores[f] = true
10                 return
11             else
12                 Counters[f] = 1
13                 Infos[f] = infos
14             end
15         else
16             if(event == "call" or event == "tail call") then
17                 Counters[f] = Counters[f] + 1
18             end
19         end
20         Inspect(event,infos)
21     end
22 end
```

Inspect is a function imported by the Inspect module and it's implemention is shown in Code 4.

Code 4: Inspect function

```
function Inspect(event,infos)
    local transfered_types = get_transfered_types(infos)
    if(event == "call" or event == "tail call") then
        push(infos)
        update_parameter_type(transfered_types,infos)
    else
        update_result_type(transfered_types)
    end
end
```

Type Categorize values into conventional types, boolean, integer, string, float, number, array, record, function An array type is defined by the union of the types of each element in the array. This result can be achieved by a map+reduce strategy, creating a new type for each element in the array and merging them by our union function.

Compatibility Matrix

Relational Metamethods

Inspect

Accessing local variables iterating getlocal for each transfered value

Hook

Basic profiler getinfo at hook events

Report

String formating

Test

Type comparison by an equality function

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>.

IERUSALIMSCHY, R. **Programming in Lua**. Fourth edition. [S.l.]: Lua.org. ISBN 978-85-903798-6-7.

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>.