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. 25p. 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. 25p. 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:

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

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

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

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

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

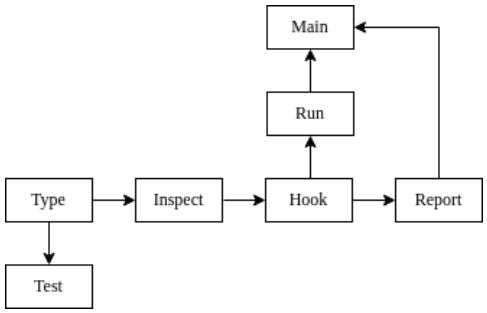

Figure 4.1: Diagram module

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.

Type extraction

The core functionality of the type extractor is the creation of a sophisticated type representation supported by the *type* function. A basic example of type extraction is shown in Code 3.

Code 3: Type Extraction

```
1  Type.new(1)
2  --> integer
3  Type.new("abc")
4  --> string
5  Type.new(true)
6  --> booelan
7  Type.new({1,2,3})
8  --> {integer}
9  Type.new({"abc", "xyz"})
10 --> {string}
11 Type.new({x = 10, y = 20})
12 --> {x:integer, y:integer}
13 Type.new({names = {"Rachel", "Derik"}, ordered = false})
14 --> {names:{string}, ordered:boolean}
```

A key aspect of the type entity is the ability to generate a new representation based on the union of other two types. It's pointed in Code 4 how some trivial types are merged together to create a new type. In the case of the union function, we override the $\_\_add$ function, as we want to use the $+$ operator to manipulate types.

Code 4: Type Union

```
-- 1
t1 = Type.new(1)
t2 = Type.new(2.0)
t1 + t2      --> float

-- 2
t1 = Type.new({x = 10, y = 20})
t2 = Type.new({x = 10, y = 20, z = 30})
t1 + t2      --> {y:integer, x:integer, z:integer?}

-- 3
t1 = Type.new({1,2,3})
t2 = Type.new({"a","b", "c"})
t1 + t2      --> {any}
```

The creation and union of types combined, enables the categorization of array types in a more elegant way. For example, we can obtain an array type by iterating over its structure, mapping each element to a new type, then reducing this array of types by folding it with our union function. The implementation of this strategy is shown in Code 5.

Code 5: Array Type Creation

```
local function map_table(tb,f)
    local result = {}
    for i = 1,#tb do
        result[i] = f(tb[i])
    end
    return result
end

local function fold_table(tb,f)
    local result = tb[1]
    for i = 2, #tb do
        result = f(result, tb[i])
    end
    return result
end

local function get_array_type(array)
    return fold_table(map_table(array, Type.new), Type.__add)
end
```

Basic profiler

The project development is inspired by a rudimentary profiler specified in Chapter 25 of the Programming in Lua book (IERUSALIMSCHY,). This profiler gives us insight of how a function can be inspected during runtime as it can be easly expanded to explore other introspective capabilities. Code 6 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 6: Hook function

```
1  function Hook (event)
2      local infos = debug.getinfo(2,"fnrSt")
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
```

Notice the invocation of *debug.getinfo* with a sequence of letters representing what information we want to obtain. The Sn part captures information about the function's name, location and source, useful when generating the desired report. The letter f fills the function value, which is used as a key value for several global tables shared across modules. The letter t tells us if the function event is part of a tail call. It's important to track this value when updating return values. Finally, the letter r fills information about the values being transferred, that is, parameters values in a call or return values in a return.

Local variables inspection

Inspect is a function imported by the Inspect module and it's implemention is shown in Code 7. It's interesting to notice that the transfered values can be inspected the same way regardless of the event type. The simplification is possible due to the ntransfer and ftransfer fields obtained before. In a call event, ftransfer is always 1, it means that the index of the first transfered value is actually the index of the first parameter, while ntransfer is the number of parameters. On the other hand, in a return event the ftransfer index is not as predictable as in a call event and the ntransfer field holds the number of transfered values in the return statement.

Code 7: Inspect function

```
function Inspect(event,infos)
    local transfered_types = get_transfered_types(infos)
    if(event == "call" or event == "tail call") then
        table.insert(Stack, infos)
        update_parameter_type(transfered_types,infos.func)
    else
        local p = table.remove(Stack)
        update_result_type(transfered_types, p.func, p.istailcall)
    end
end
```

Another instrospective function explored by our extractor is the *debug.getlocal* function which is is responsible for accessing local variables within a closure. Togheter with the transfered value indexes, *getlocal* can to access the parameter and return values we want and extract it's type. This logic is shown in Code 8.

Code 8: Get Transfered Types

```
local function get_transfered_types(infos)
    local t = {}
    for i=infos.ftransfer,(infos.ftransfer + infos.ntransfer) - 1 do
        local _, value = debug.getlocal(4,i)
        table.insert(t, Type.new(value))
    end
    return t
end
```

Lua functions feature a concept called *proper tail calls*, meaning that the calling function does not have its respective space in the stack after a tail call is made. Lua's proper tail call design is appropriate for many programming situations, but it requires our extarctor an extra step for updating return values

of functions. Because Lua does not keep any information of the calling function of a tail call, our extractor still had to keep this information for updating the return values appropriately. Code 9 shows that, by manipulating a stack table, we can maintain the syncronism between the return type inspection and the program's call stack.

Code 9: Update Return Types

```
local function update_return_type(types, func, istailcall)
    add_return_types(types,func)
    while istailcall do
        local p = table.remove(Stack)
        func = p.func
        istailcall = p.istailcall
        add_return_types(types,func)
    end
end
```

Type comparison

In order to test if our type representation is correct, we explored the relational metamethod ___*eq*, giving a new meaning under our type context. Two types are equal if it has the same type representation. With that in mind, an equality function can be defined recursevly, by analysing each type tag and comparing subsequent structures if needed. An example of type comparison is shown in Code 8.

Code 10: Type Comparison

```
INTEGER = {tag = "integer"}
Type.new(1) == INTEGER  --> true

EMPTY = {tag = "empty"}
Type.new({}) == EMPTY   --> true

ANY = {tag = "any"}
Type.new("abc") == ANY  --> false


BOOLEAN = {tag = "boolean"}
BOOLEAN_ARRAY = {tag = "array", arrayType = BOOLEAN}
Type.new({true,false}) == BOOLEAN_ARRAY --> true

FLOAT = {tag = "float"}
FLOAT_RECORD = {tag = "record", recordType = {x = FLOAT}}
Type.new({x = 3.14}) == FLOAT_RECORD    --> true
```

6
Results

In this section, we will give some examples of generated reports obtained by passing a full program as input to the type extractor. The program analysis is not supposed to be fast. The overhead of a Lua call for each hook is high enough to leverage time performance. In this section we will present some reports generated by the extractor when analysing simple Lua programs and also Lua benchmark programs.

Basic

A basic result of the extractor is represented by the Image 6.1. The output generated is related to the execution of Code 11. Image 6.2 shows the output for Code 12. In this example, regardless of the input type, because the result generated by *type* is always an string.

Code 11: Basic example 1

```
1 function foo(n)
2     return math.sqrt(n)
3 end
4 foo(1)
```

Code 12: Basic example 2

```
1 function foo(a)
2     return type(a)
3 end
4 foo({x = 10, y = 20, p = {3.0,5.0}})
```

```
============================================================
Number of functions:    1
Total calls:    1
============================================================
[exemplos/basic_1.lua]:1 foo    (integer)->(float)      1
============================================================
```

Figure 6.1: Basic example 1

```
================================================================================
Number of functions:    1
Total calls:    1
================================================================================
[exemplos/basic_2.lua]:1 foo    ({x:integer, y:integer, p:{float}})->(string)   1
================================================================================
```

Figure 6.2: Basic example 2

```
============================================================
Number of functions:    1
Total calls:    1
============================================================
[exemplos/basic_3.lua]:1 foo    ({integer})->(nil)      1
============================================================
```

Figure 6.3: Basic example 3

```
================================================
Number of functions:    1
Total calls:    2
================================================
[exemplos/basic_4.lua]:1 foo    (any)->(any)    2
================================================
```

Figure 6.4: Basic example 4

Another useful example is shown by the output of Code 13 and its output in Image 6.3. Instead of analysing transfered values only in the return event, the extractor analyses the parameter type in a call event, before any other assignment is made, preserving the type of the variables before the function's execution.

Code 13: Basic example 3

```
1 function foo(a)
2     a = nil
3     return a
4 end
5
6 foo({1,2,3})
```

Image 6.4 exeplifies a union of function types, generated by the execution of Code 14. An union between an array of integer and a boolean type results in a dynamic type.

Code 14: Basic example 4

```
1 function foo(a)
2     return a
3 end
4 foo({1,2,3})
5 foo(true)
```

Finally, Image 6.5 shows the handling of proper tail calls in action. Code 15 has two functions, where the last statement of *foo* is a call to *boo*, when the return event of this function is triggered, the return type of the calling function is updated correctly.

```
============================================================
Number of functions:    2
Total calls:    2
============================================================
[exemplos/basic_5.lua]:4 foo    (nil)->(integer)        1
[exemplos/basic_5.lua]:1 boo    (nil)->(integer)        1
============================================================
```

Figure 6.5: Basic example 5

```
======================================================================
Number of functions:    1
Total calls:    172233
======================================================================
[exemplos/ack.lua]:4 Ack        (integer,integer)->(integer)    172233
======================================================================
```

Figure 6.6: Ackerman output

Code 15: Basic example 5

```
1 function boo()
2     return 1
3 end
4 function foo()
5     return boo()
6 end
7 foo()
```

Ackerman

The Ackerman function is known in the community by its increasing execution time. We can observe the output of the extractor when passing an Ackerman script as input in Image 6.6. In this example, ackerman was called with the parameters (3,6) and had 172233 executions.

Some limitations

Although a more refiend type representation is implemented by the type extractor, there are some limitations we must adress. For example, when dealing with a recursive structure, our tool is not capable of identifing it's recursiveness, leading to an extensive type representation. Code 16 shows a bad and a nice way to represent recursive types.

Code 16: Recursive representation

```
-- bad way (extractor)
list:
    { element:integer,
      next:
        { element:integer,
          next:
            { element:integer,
              next:
                { ... }
            }
        }
     }

-- nice way
list: {element:integer, next:list}
```

7
Final Considerations

The motivation of this project is to help Lua programmers to understand and optimize dynamically typed code. Here we described an extraction tool for Lua programs that is capable of reporting the program's function types in an friendly way. A type extraction tool is useful for dynamically typed code, as it provides information extracted during runtime, but inspecting a Lua program is a challenging task as we must deal with proper tail calls, first class functions and associative arrays. The information described in this document serves as an stimulus for understanding the relationship bewtween statically typed languages and dinamically typed ones. It also poses as a contribuition for the Lua community as a complementary tool for code profiling and inspection. The source code as well as a brief documentation is available at https://github.com/fvcortes/TypeExtractor

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>.