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

To my parents, for their support
and encouragement.

Table of contents

List of figures

List of tables

List of algorithms

List of codes

List of Abreviations

ADI – Análise Digital de Imagens
BIF – *Banded Iron Formation*

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 also generally has better performance because compile-time type information helps generating optimized machine code. However, programmers are left empty handed when faced with re-writing code from a dynamically typed language to a statically typed one. Most of the times they have to re-write the entire system if gradually typed languages are not an option.
In order to aid programmers to understand the relationship of the types between parts of a program we decided to build a dynamic type extractor for the Lua programming language. The Type Extractor will give you information about function types in a program inspected during runtime. With this information programmers can more easly migrate to a statically typed language. The information generated by our Type Extractor can also be used for code inspection and serve as an useful documentation.
This software is destinated for programmers who wants to understand the existing types within their Lua programs, developers aiming to produce a useful documentation and also to inspect the type behaviour of variables and functions present in their code.

2
Project Scope

The extractor can analyse each function call and return by using reflection properties of the Lua programming language. With the debug library, we ca n register hook functions to inspect the behaviour of a program's execution and then compute the types of functions and variables present in the code .
The Type Extractor can be used by two approaches.

- Full Analysis: A full program analysis can be made by passing a Lua program as input to the extractor. In this approach, each possible fun ction 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 sel ect what part of the program they want to analyse.

In the end of each execution, a report will be generated. This report has information about parameters and return types of each analysed function.
These usage scenarios enables the extractor to be used as an auxiliary tool for migrating from dynamically to statically typed languages. It also ser ves as a good documentation for functions parameter and return types. Giving tools for understanding the type relations inside a program helps progra mmers to debug and optimize dynamically typed code.

3
Proposal

Equation example 1:

$$\begin{aligned} \min_{u} \int_{x_i \in X} \int_{x_j \in X} q_{ij} u_i u_j dada + \int_{x_i \in X} ||x' - x_i|| u_i da \\ s.t. \quad u \in [0,1] \quad \wedge \quad \int_{x_i \in X} uda = a_0, \end{aligned} \tag{3-1}$$

Equation exmaple 2:

$$\begin{aligned} \min_{\mathbf{u}} \alpha \mathbf{u}^T \mathbf{A}^T \mathbf{Q} \mathbf{A} \mathbf{u} + \beta \mathbf{d}^T a' \mathbf{A} \mathbf{u} + \gamma \mathbf{u}^T \mathbf{G}^T \mathbf{G} \mathbf{u} + \delta \mathbf{f}^T a' \mathbf{A} \mathbf{u} \\ s.t. \quad \mathbf{0} \leq \mathbf{u} \leq \mathbf{1} \wedge \mathbf{a}^T \mathbf{u} = a_0. \end{aligned} \tag{3-2}$$

Equation example 3:

$$\mathbf{G} = (g_{ij}) = \begin{cases} \sum_{f_k \in N_f(f_i)} l_{ik} & i = j \\ -l_{ij} & e_{ij} \in E \\ 0 & \text{otherwise} \end{cases} \tag{3-3}$$

Code 1: Mean Filter

```
#
    ----------------------------------------------------------------------#

# Create filter function
# l is the width of window
#
    ----------------------------------------------------------------------#

meanfilter <- function( l, imagem ) {
  if( l%%2 == 0 )
    print("Please, type an odd number!")
  imagem.result <- imagem
  lp1d2 <- (l-1)/2
  L <- dim(imagem)[1]
  C <- dim(imagem)[2]
  for( j in as.integer(lp1d2+1) : as.integer(C-lp1d2)) {
    for( i in as.integer(lp1d2+1) : as.integer(L-lp1d2)) {
      imagem.result[i,j] <- mean(imagem[as.integer(i-lp1d2):as.
          integer(i+lp1d2), as.integer(j-lp1d2):as.integer(j+lp1d2)
          ])
    }
  }
```

```
17   print("Image filtred with success!")
18   return(imagem.result)
19 }
20 #
       ------------------------------------------------------------------------#

21 # End of Script.
22 #
       ------------------------------------------------------------------------#
```

Algorithm 1: Escolha das amostras inicias

Input: Malha e quantidade de pontos a ser amostrado
Output: Pontos amostrados na malha

1 *Crie um vetor de números randômicos entre* $[0, 1]$ *com a quantidade de pontos a ser amostrada e ordene-o*
2 *Calcule a área total dos triângulos da malha*
3 **for** $i = 0$ **to** *numeroDePontos* **do**
4 *Navegue entre as faces acumulando a sua* $\frac{area}{areaTotal}$ *até achar a face com valor acumulado* $\geqslant$ *numerosRandomicos[i]*
5 *Pegue um ponto randômico dentro da face utilizando o método de Turk e adicione no vetor do resultado*

4
Results

Table example. Table 4.1 shows results.

Table 4.1: Results for devil mesh

	Mean Vertex Distance	L2 Vertex Based	Mean Quadric	MSAE	L2 Normal Based	Tangential	Mean Discrete Curvature	Area Error	Volume Error
(??)	0.061277	0.110973	0.236219	19.697900	0.055170	0.047678	0.090284	0.051443	0.045645
(??)	0.001293	0.002800	0.002289	21.237300	0.021589	0.013026	0.087991	0.000364	0.002621
(??)	0.001439	0.002880	0.003540	14.043200	0.012654	0.008911	0.055849	0.007806	0.000582
(??)	0.000713	0.001537	0.001824	12.171400	0.009654	0.005781	0.054567	0.005617	0.000425
(??)	0.002531	0.004560	0.007108	13.830100	0.017459	0.010314	0.114528	0.001686	0.001786
(??)	0.001623	0.003079	0.005048	10.454200	0.015233	0.008054	0.094668	0.002629	0.001326
(??)	0.000737	0.001548	0.001493	16.880800	0.014129	0.006974	0.079952	0.000209	0.002375
Ours	0.000987	0.001902	0.002686	11.574200	0.010632	0.006796	0.075106	0.003970	0.000722

4.1
Comparison

5
Conclusion and future work

We proposed an algorithm for triangular mesh denoising with detail preservation...

Code 2: Mean Filter

```
#
    ------------------------------------------------------------------------#

# Create filter function
# l is the width of window
#
    ------------------------------------------------------------------------#

meanfilter <- function( l, imagem ) {
  if( l%%2 == 0 )
    print("Please, type an odd number!")
  imagem.result <- imagem
  lp1d2 <- (l-1)/2
  L <- dim(imagem)[1]
  C <- dim(imagem)[2]
  for( j in as.integer(lp1d2+1) : as.integer(C-lp1d2)) {
    for( i in as.integer(lp1d2+1) : as.integer(L-lp1d2)) {
      imagem.result[i,j] <- mean(imagem[as.integer(i-lp1d2):as.
          integer(i+lp1d2), as.integer(j-lp1d2):as.integer(j+lp1d2)
          ])
    }
  }
  print("Image filtred with success!")
  return(imagem.result)
}
#
    ------------------------------------------------------------------------#

# End of Script.
#
    ------------------------------------------------------------------------#
```

6
Bibliography

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>.