“Learning Highly Recursive Input Grammars”: A Replication Study

Sulav Malla

A thesis presented in partial fulfilment of the requirements for the degree of

Bachelor of Engineering Honours (Software)

Supervisors:
Rahul Gopinath, The University of Sydney

School of Electrical and Computer Engineering
The University of Sydney

November 04, 2025

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus

commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.1

Contents

Chapter 1

INTRODUCTION

In the field of software testing, generating test inputs for a program (fuzzing) is a well-known and popular technique. To improve the effectiveness of fuzzers, recent research has focused on recovering input grammars from existing programs, as incorporating knowledge about the input language and grammar of the program under test drastically improves the effectiveness of fuzzers [1].

Learning Highly Recursive Input Grammars, authored by Lemieux C., Sen K., and Kulkarni N., and published in 2021 at UCB [2], presents an algorithm called ARVADA, developed for this purpose. ARVADA attempts to learn context-free grammars (CFGs) of a specified program given a set of valid inputs and a boolean-value black-box oracle $\mathcal{O}$. Along with presenting the algorithm, the paper evaluates it by comparing ARVADA to GLADE [3], a previously developed state-of-the-art algorithm for the same task in a similar setting. During the evaluation, it was observed that the F1 scores of GLADE were much lower than those reported in the official GLADE paper [3]. This led to a replication study of the original GLADE paper, conducted by Gopinath R., Bachir B., and Zeller A., and published as "Synthesizing Input Grammars: A Replication Study" [4] at CISPA, which investigated the accuracy of the results in the original paper. Similarly, as done by the researchers at CISPA, this thesis aims to replicate ARVADA to reproduce and investigate the results presented in the original paper [2].

This thesis is an attempt to reproduce ARVADA in a clean-room environment, meaning with no reference to or knowledge of the original implementation, but only the abstraction and explanations provided in the paper [2]. The implementation language of choice is C.

First, this paper will introduce general concepts of grammar and parsing, the importance of learning input grammars, a deeper explanation of ARVADA, and the motivation for conducting a replication study. This is followed by a brief explanation of ARVADA and how it works according to the original paper [2], then a more in-depth explanation of the algorithm and how it was reproduced in C with reference to the code. Afterward, an evaluation compares the results of the reproduced implementation with those from the original paper [2]. Finally, a discussion highlights and comments on the original algorithm, the reproduced results, and the overall process of conducting the replication study, followed by a brief conclusion.

- All code and implementation are open source and can be found here: `https://github.com/Stainima/ARVADA`

Chapter 2

GENERAL REVIEW OF LITERATURE

2.0.1 What is ARVADA?

Introduction

ARVADA is an algorithm published in "Learning Highly Recursive Input Grammars" [2] at the University of California, Berkeley in 2021. It is designed to learn context-free grammars from a set of positive examples and a Boolean-valued oracle $\mathcal{O}$. Starting from initially flat parse trees, ARVADA repeatedly applies two specialized operations, **bubbling** and **merging**, to incrementally add structure to these trees. From this structured representation, it extracts the smallest possible set of context-free grammar rules that accommodate all the given examples. The algorithm aims to generalize the language as much as possible without overgeneralizing beyond what is accepted by $\mathcal{O}$.

Like GLADE [3], ARVADA operates under the assumption of a black-box oracle $\mathcal{O}$. This means that ARVADA has no access to or knowledge of the internal workings of the oracle and can only observe the Boolean values returned by $\mathcal{O}$.

Explanation

ARVADA takes as input the oracle $\mathcal{O}$ and a set of positive, valid oracle inputs S. For each string $s \in S$, querying $\mathcal{O}(s)$ returns `True`. The algorithm begins by constructing a flat parse tree for each string in S. Each tree has a single root node t_0 whose children correspond to the individual characters of the input string s.

Next, ARVADA performs the **bubbling** operation. In this step, a sequence of sibling nodes in the tree is selected and replaced with a new non-terminal node. This new node takes the selected sibling nodes as its children, thereby introducing an additional level of structure. Essentially, ARVADA transforms sequences of terminal nodes in the flat parse tree into subtrees by introducing new non-terminal nodes and progressively adding structure to the tree.

ARVADA then decides whether to accept or reject each bubble by checking whether the newly bubbled structure enables a sound generalization of the learned grammar. Each non-leaf node in the tree can be viewed as a non-terminal in the emerging grammar. To determine whether a bubble should be accepted, ARVADA checks whether replacing any node in the tree with the new bubbled subtree results in the generation of valid input strings according to $\mathcal{O}$. If the replacement produces valid strings, the bubble is accepted, and the tree is restructured so that both the bubbled subtree and the replaced node share the same non-terminal label.

The addition of new non-terminal nodes expands the language defined by the learned grammar, since any string derivable from the same label can now be substituted interchangeably. This relabeling of the bubbled subtree and the replaced node is called a **merge**, as it merges the labels of two previously distinct nodes in the tree. If a bubble is not accepted, it is discarded, and none of the trees are affected or structurally modified.

Walkthrough

2.0.2 What is GLADE?

GLADE is an algorithm proposed by Bastani et al., published in Synthesizing Input Grammars at PLDI 2017 [3]. Like ARVADA, GLADE uses a set of valid inputs and black-box access to the program, with the aim of automatically approximating the context-free input grammar of the given program.

Because GLADE and ARVADA share a similar experimental setting, GLADE was used as a benchmark for ARVADA during its evaluation. It was shown that GLADE, on average, had a faster runtime compared to ARVADA. However, ARVADA outperformed GLADE in terms of generalization across the 11 benchmarks, achieving a higher F1 score on 9 of the 11 benchmarks.

2.0.3 What are Grammars?

Grammars in computer science can be defined as a set of rules by which valid sentences in a language are constructed [5], serving as a blueprint for the language. Beginning with a start symbol, which is a single non-terminal, production rules are applied sequentially, adding symbols from the alphabet according to the grammar's production rules, to generate a string that is valid in the language [6].

A grammar can be represented by the tuple $< N, T, P, S >$, where:

- N is a finite, non-empty set of non-terminal symbols.
- T is a finite set of terminal symbols.
- P is a finite set of production rules.
- S is the start symbol (a non-terminal symbol).

Here, the production rules P define all symbol substitutions that can be performed recursively to generate different symbol sequences [6].

2.0.4 What are context free grammars?

2.0.5 Further Quesitons

What are parsers?

Why do a replication study?

2.0.6 Why C?

In the original study [2], the ARVADA algorithm was implemented in Python. When compared to GLADE [3], which was implemented in Java, ARVADA exhibited a slower average runtime across all benchmarks. In the study, this was attributed to the natural runtime disadvantage of Python compared to Java, which is valid, however the possibility that ARVADA itself might be inherently slow was not acknowledged.

In a comparative study, A Pragmatic Comparison of Four Different Programming Languages [7], it was found that if speed and efficiency are important, C is a better option than Python. C, being a mid-level, statically typed, structured language that runs under a compiler, is consistently faster than dynamic languages that run under an interpreter, such as Python [8]. In addition to being a structured language, C also provides only essential features. These limited features contribute to its efficiency but also introduce a higher level of complexity compared to Python [7], [8].

Therefore, with the aim of investigating and potentially improving the runtime bottleneck, C was chosen as the language of implementation in this replication study. The increase in implementation complexity due to the nature of programming in C compared to Python was also acknowledged.

Why ARVADA / Problem Statement?

Why is learning highly input grammar important?

What are other Similar works done?

What is the work done in this field after ARVADA?

Chapter 3

METHODOLGIES & IMPLEMENTATION

Methodolgies: ARVADA walkthrough

How you did it, and point out any differences?

Listing 3.1: Struct used in code to store all *trees*

```c
typedef struct nodes{
    int capacity;
    int count;
    struct node **nodes;
} Nodes;
```

Listing 3.2: Struct used in *trees*

```c
typedef struct node{
    int capacity;
    char character;
    struct node *parent;
    int t;
    int num_child;
    int pos;
    struct node **children;
} Node;
```

Chapter 4

EVALUATION

Chapter 5

DISCUSSION

Chapter 6

CONCLUSION

References

[1] R. Gopinath and A. Zeller. "Building Fast Fuzzers," pre-published.

[2] N. Kulkarni, C. Lemieux, and K. Sen, "Learning Highly Recursive Input Grammars," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Melbourne, Australia: IEEE, Nov. 2021, pp. 456–467, ISBN: 978-1-6654-0337-5. DOI: 10.1109/ASE51524.2021.9678879.

[3] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing Program Input Grammars,"

[4] B. Bendrissou, R. Gopinath, and A. Zeller, ""Synthesizing input grammars": A replication study," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, San Diego CA USA: ACM, Jun. 9, 2022, pp. 260–268, ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523716.

[5] T. Jiang, M. Li, B. Ravikumar, and K. W. Regan, "Formal Grammars and Languages,"

[6] "Grammar in Theory of Computation," Grammar in Theory of Computation.

[7] S. Ali and S. Qayyum. "A Pragmatic Comparison of Four Different Programming Languages," pre-published.

[8] R. Kumar, S. Chander, and M. Chahal, "Python versus C Language: A Comparison," vol. 9, 2022.

[9] M. Hendriks and V. Zaytsev, "Consider it Parsed!" Thesis, University of Twente, Enschede, Netherlands.

[10] M. Schröder, J. Cito, and T. U. Wien, "Static Inference of Regular Grammars for Ad Hoc Parsers,"