June 21, 2024

TU Delft

Splitting Context-Free Grammars to Optimize Program Synthesis

Herb.jl

Responsible Professor
Sebastijan Dumančić

Dennis Heijmans | D.Heijmans@student.tudelft.nl

Supervisor
Reuben Gardos Reid

1. Introduction

- Programming synthesis [3] is the task of searching for a program satisfying a set of examples.
- The search space of this problem consist of all possible programs that can be constructed from a given context-free grammar.
- Enumerative programming synthesis quickly becomes intractable.
- Divide and conquer techniques [1,2] on the set of examples has been proven to be effective.
- Can a similar strategy be applied to a problem's context-free grammar?

How can an arbitrary context-free grammar be split in subgrammars that can make the synthesis of programs more efficient?

1. How can a context-free grammar be split into smaller subsets?
2. How to learn a program from a set of subgrammars?
3. How to determine which subgrammars to combine to find a solution?

2. Methodology

Splitting Grammar

1. $Element \rightarrow Number$
2. $Element \rightarrow Bool$
3. $Number \rightarrow 1 \mid 2 \mid 3 \mid x$
4. $Number \rightarrow Number + Number$
5. $Bool \rightarrow Number = Number$
6. $Number \rightarrow \text{if } Bool \text{ then } Number \text{ else } Number$
7. $Bool \rightarrow \text{if } Bool \text{ then } Bool \text{ else } Bool$

Figure 1: Simple context-free grammar

- A problem's context-free grammar is converted into a dependency graph.

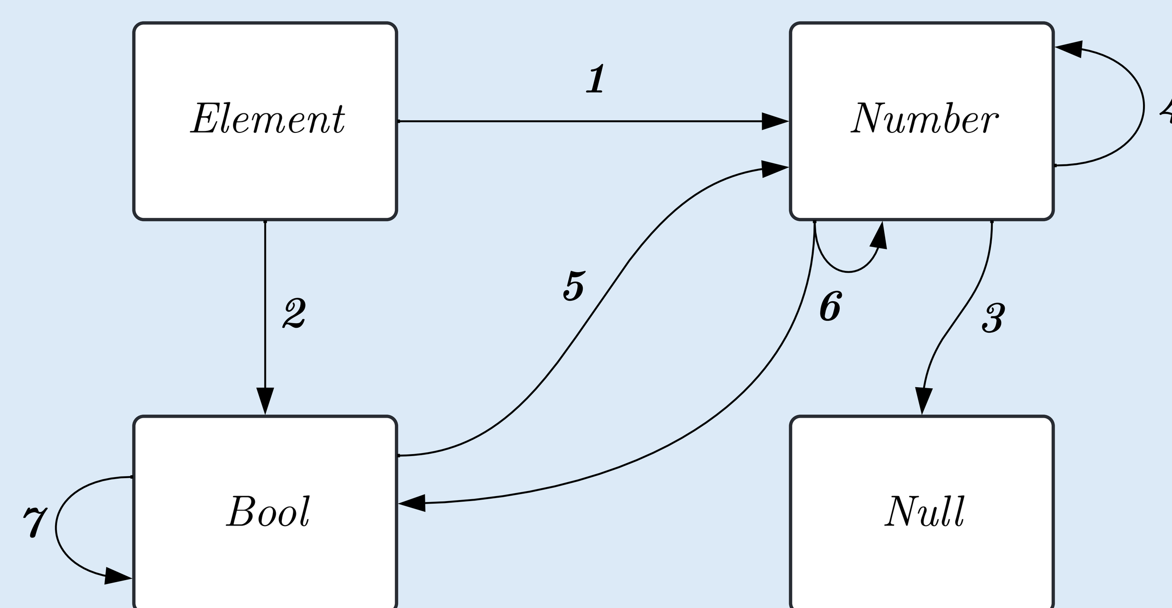

Figure 2: Dependency graph

- Each node stands for a symbol, and every edge represents a set of rules.
- For every rule, a subgrammar is created by finding the shortest path from the starting rule to a set of terminals by travelling through the edge of that rule.
- Grammars that are subsets or duplicates of others are not considered, otherwise their programs will be run multiple times.

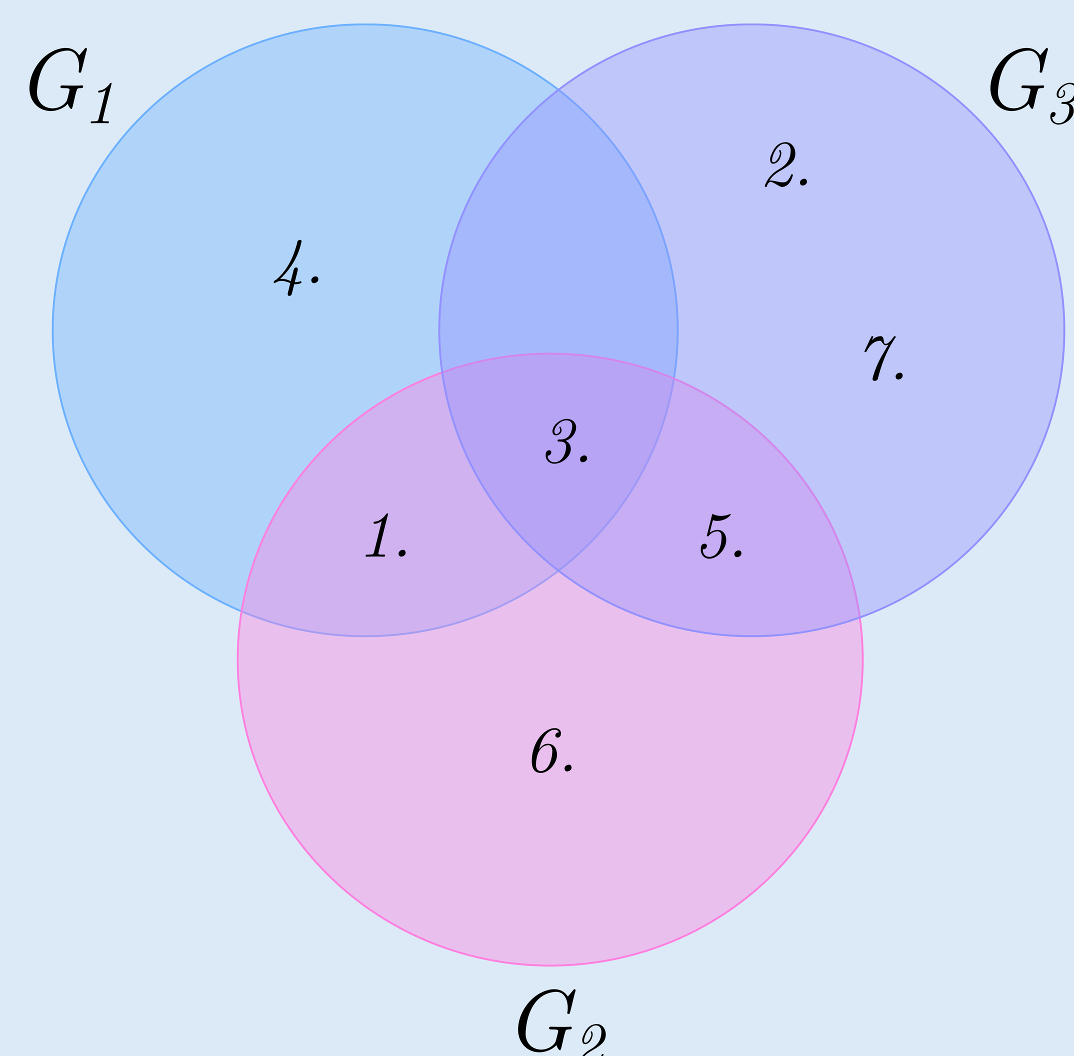

Figure 2: Generated subgrammars

Exploring Subgrammars

- All possible combinations of n grammars will be assigned a score from 0 to 1 based on how many examples programs generated by them can solve.

Exploiting Subgrammars

- Each grammar receives a fraction of the enumerations based on its score divided by the sum of all scores.

3. Experimental Setup and Results

- The program iterator is implemented in HerbSearch.jl[1].
- This repository is part of Herb.jl[2] which is a program synthesis library written in Julia.
- Our `GrammarSplittingIterator` will be compared against the `BFSIterator`.
- HerbBenchmarks.jl[3] provides us with a collection of benchmarks for testing the iterators.
- We will use the PBE SLIA Track 2019 from the SyGuS competition, which includes 100 string-manipulation problems
- Our iterator will use the Levenshteinedit distance metric to give grammars partial points, as most examples consist of string outputs.
- The grammars from the 100 benchmarks have 27 rules on average and each consistently split into either 8 or 9 subgrammars using our iterator.
- Both iterators are allowed a maximum of 10 million program enumerations per problem.
- The `GrammarSplittingIterator` will run with $n = 3$ and 5% of the enumerations will be used for exploration.

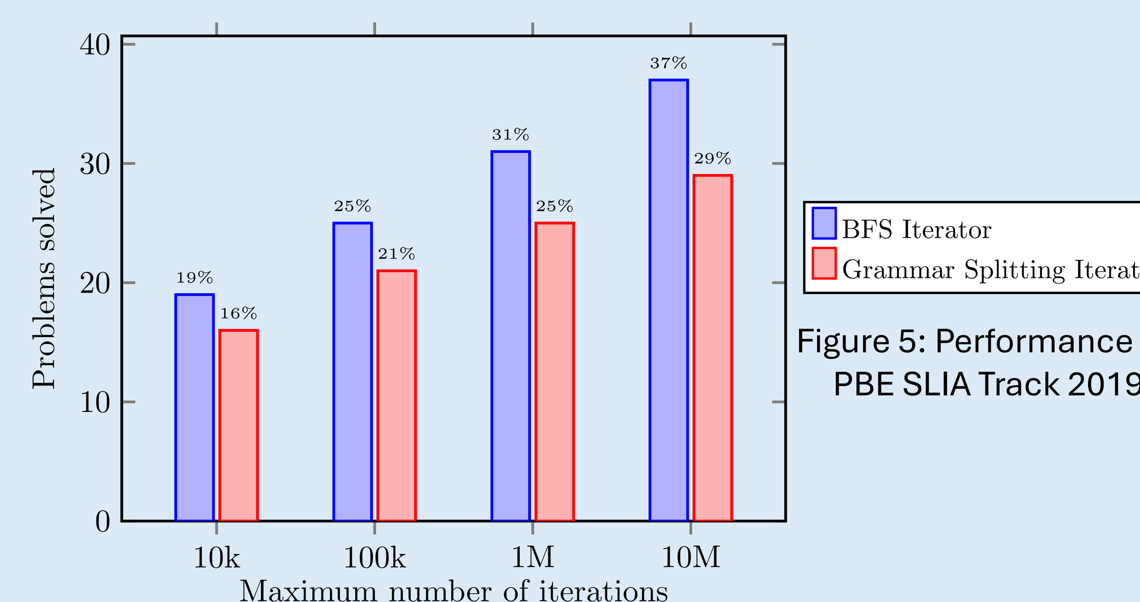

Figure 5: Performance on PBE SLIA Track 2019

4. Discussion

- The `GrammarSplittingIterator` uses on average 7 times more iterations than the `BFSIterator`.
- In the medium sized problems where the solution is synthesized in the exploitation phase, our iterator performs better when the right context-free grammar is run first.
- It was able to synthesize a program in 0.38 times the enumerations compared to BFS.

5. Responsible Research

- This experiment can be reproduced as the iterator as well as the benchmarks are all in the open-source Herb.jl framework.

6. Conclusion

- The `GrammarSplittingIterator` performs worse than `BFSIterator`.
- Running it in parallel could potentially let our iterator exceed the `BFSIterator`.

7. Future Work

- Make the `GrammarSplittingIterator` run in parallel.
- Add constraints that prevent duplicate programs as much as possible
- Continue experimenting with different merging strategies

8. References

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling Enumerative Program Synthesis via Divide and Conquer. In Axel Legay and Tiziana Margaria, editors, Tools and Algorithms for the Construction and Analysis of Systems, volume 10205, pages 319–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017. ISBN 978-3-662-54576-8 978-3-662-54577-5. doi: 10.1007/978-3-662-54577-5 18. URL https://link.springer.com/10.1007/978-3-662-54577-5 18. Series Title: Lecture Notes in Computer Science.

[2] Andrew Cropper. Learning Logic Programs Though Divide, Constrain, and Conquer. Proceedings of the AAAI Conference on Artificial Intelligence, 36(6):6446–6453, June 2022. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v36i6.20596. URL https://ojs.aaai.org/index.php/AAAI/article/view/20596.

[3] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. Number 4.2017, 1-2 in Foundations and trends in programming languages. Now Publishers, Hanover, MA Delft, 2017. ISBN 978-1-68083-292-1

[4] Gonzalo Navarro. A guided tour to approximate string matching. ACM Computing Surveys, 33(1):31–88, March 2001. ISSN 0360-0300, 1557-7341. doi: 10.1145/375360.375365. URL https://dl.acm.org/doi/10.1145/375360.375365

[1] https://github.com/Herb-AI/HerbSearch.jl
[2] https://herb-ai.github.io/
[3] https://github.com/Herb-AI/HerbBenchmarks.jl