Splitting Context-Free Grammars to Optimize Program Synthesis[*]

Dennis Heijmans
Supervisors: Sebastijan Dumančić, Reuben Gardos Reid
EEMCS, Delft University of Technology, The Netherlands

Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2022

Abstract

Program synthesis is the task of generating a program that suffices the intent of a user based on a set of input-output examples. Searching over the set of all possible programs becomes intractable very quickly. Therefore, divide and conquer techniques have become popular within the field, but have mainly been applied on the set of examples. However, this paper focuses on applying the strategy on the problem's context-free grammar by splitting it into subgrammars.

Our new technique first splits the grammar by making a dependency graph showing how all rules relate the different types of symbols. Afterwards, there is an exploration and exploitation phase where different sets of subgrammars will be given a score and get allocated an amount of enumerations to generate programs based on that score.

The new technique is implemented as an iterator in Herb.jl which is a program synthesis framework. The iterator is then benchmarked against a plain BFS iterator using 100 string-manipulation problems. The grammar splitting strategy needs on average more enumerations to find a program solving all examples compared to the BFS iterator. However, running the different grammars from the iterator in parallel could allow the iterator to find a solution from one of the grammars earlier.

Introduction

Program Synthesis [3] is considered the holy grail within the field of Computer Science. It is the task of automatically creating a program in an underlying language that satisfies the user intent expressed in the form of some specification. More specifically, the synthesis process consists of a search over the space of all possible programs within a given language with the aim of finding a program that satisfies a set of input-output examples.

Take for example the following list of inputs that has to be mapped to a list of outputs:

$$[1, 2, 3] \to [2, 4, 8]$$

[*]An electronic version of this thesis is available at https://repository.tudelft.nl/

The job of a program synthesizer would then be to enumerate a lot of different programs from a context-free grammar to find one that solves all three examples. A possible solution covering all the examples in this case could be $f(x) = 2^x$.

As these search spaces tend to be extremely big if not infinitely large, finding solutions quickly becomes intractable. Therefore, clever techniques have been developed to make synthesizing programs more feasible.

Multiple divide and conquer techniques [1, 2] have been developed already to cut down the search space. However, trying to apply this dividing strategy on the context-free grammar of a problem has not been done before. Therefore, a research question from this knowledge gap arises:

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

To tackle this research question a couple of sub questions need to be answered first:

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

In this paper a new program synthesis technique using subgrammars will be explained. It will use a divide and conquer strategy as it is based around the idea of splitting and combining grammars. The goal is to optimize the amount of program enumerations it takes to synthesize a solutions to a program synthesis problem.

Background

EUSolver [1] already uses divide and conquer but applies this idea on splitting the set of examples. It generates predicates from the problems context-free grammar and uses a decision tree to group the examples by these predicates. Afterwards, single programs that solve smaller groups can be combined using a structure of if-statements. There also exist another strategy that uses decision trees, but combines this also with machine learning and constraint-driven search [2].

Methodology

This section outlines a solver for addressing program synthesis problems while making use of subgrammars. A problem presented to this solver must include a collection of input-output examples along a context-free grammar to construct programs from. Initially, the solver will split the context-free grammar into multiple subgrammars. Subsequently, it explores and evaluates various combinations of these subgrammars. Finally, the grammars are exploited based on their scores. Throughout this section, an example will be used to clarify each step thoroughly.

Splitting Grammar

Every program synthesis problem provides us with a set of examples together with a context-free grammar. This grammar contains a set of rules for constructing a valid program. Consider the grammar from figure 1. Its rules can be used to construct simple Boolean and arithmetic expressions. The starting symbol for this grammar is $Element$, as it will be the root node for all abstract syntax trees generated from this grammar.

1. Element → Number
2. Element → Bool
3. Number → 1 / 2 / 3 / x
4. Number → Number + Number
5. Bool → Number = Number
6. Number → if Bool then Number else Number
7. Bool → if Bool then Bool else Bool

Figure 1: Simple context-free grammar

In many cases, not all rules are needed to generate a program that solves the set of examples from the problem. Therefore, it might be helpful to generate programs from several subsets of these rules rather than from all rules at once.

To generate these so-called subgrammars, we must split our initial context-free grammar. Since we cannot determine in advance which rules are required to produce a program that meets all examples, every rule from the initial grammar should appear in at least one of the subgrammars. Additionally, a subgrammar is only meaningful if all its rules can be used to generate a valid program. For that reason, all rules should be reachable from the starting symbol and all non-terminals in these rules should be substitutable such that no non-terminals are left behind.

To establish these meaningful subgrammars, we will construct a dependency graph of our initial grammar. In this graph, each symbol from the grammar is represented by a node, and the edges between them show how the rules connect these symbols. For instance, in figure 2, rule number 6 has the symbol $Number$ on its left-hand side and generates a string containing the $Boolean$ and Number symbol.

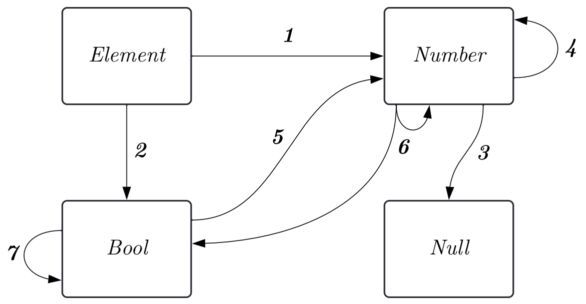

Figure 2: Dependency graph

Consequently, the graph has an edge from $Number$ pointing to both $Number$ and $Boolean$. All edges that represent terminal rules point to $Null$, as they do not depend on any non-terminals.

Once we have a dependency graph finished, for every rule in the initial grammar we will generate the smallest possible grammar in which that rule is contained. We use breadth-first search to identify the shortest path from the starting symbol to $Null$ passing through the desired rule. For example, to create a grammar that includes rule number 6, we need to include rule 1 to reach this rule from the starting symbol, and rules 3 and 5 to replace all its non-terminals with terminals.

1.[1,3]
2.[2,5,3]
3.[1,3]
4.[1,4,3] $\longrightarrow$ *[1,4,3], [1,6,5,3], [2,7,5,3]*
5.[2,5,3]
6.[1,6,5,3]
7.[2,7,5,3]

Figure 3: Pruning subgrammars

Since the dependency graph in our example has seven edges, we end up with the seven subgrammars. Some of these grammars are subsets of others. Checking all of these would lead to executing the same programs multiple times. Therefore, we disregard any subgrammar that is a proper subset or duplicate of another. For our example, this step is shown in figure 3. Finally, to visualize the subgrammars left after this pruning step, figure 4 shows a Venn diagram of these grammars.

Exploring Subgrammars

Now that we have broken our initial grammar down into its core subgrammars, we need to determine which sets of subgrammars are most promising for synthesizing our program. Since our solver does not know the meaning of the rules from each subgrammar, we will treat them equally during this exploration process. It is very unlikely that only one subgrammar will be able to synthesize a program satisfying the full problem, especially when working with larger grammars rather than our small example. Therefore, we will first look over all possible combinations of n grammars and

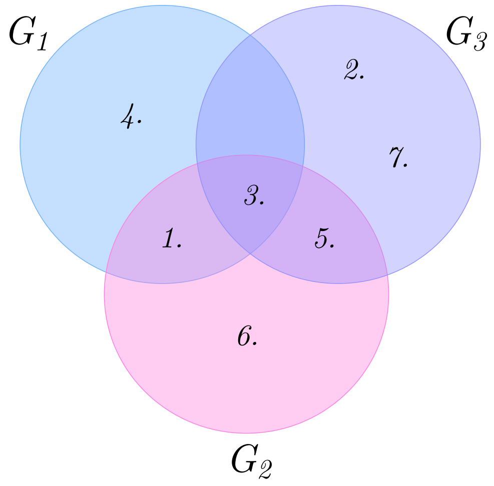

Figure 4: Venn diagram of subgrammars

assign each combination a score from 0 to 1. The value of n will be parameter for our solver and must be chosen by the user.

In our example, with only three subgrammars, we will select $n = 2$ to illustrate the process. There are $\binom{3}{2} = 3$ different combinations of two grammars possible from our set of three grammars. This means we will have to explore the grammars $G_1 \cup G_2$, $G_1 \cup G_3$, and $G_2 \cup G_3$.

To assign all combinations of grammars a meaningful score, we need a metric that effectively represents the importance of the grammars. For that reason, we will enumerate a fixed number of programs from each grammar and check what the highest percentage of examples is that a program generated by it can cover. It is possible that a program solving all examples will be already found during this phase of exploration. In that case, there is no need to explore further, and we have solved our problem.

Exploiting Subgrammars

After having assigned a score to all candidate grammars, we proceed by using the remaining enumerations to exploit them accordingly. Each grammar receives a fraction of enumerations based on its score divided by the sum of all scores. For example, if one grammar has a score of 0.7, and the other two grammars have scores of 0.3 and 0.6, we allocate $\frac{0.7}{0.7+0.3+0.6} = 43.75\%$ of our enumerations to this grammar. This final step of the process concludes when a solution is found, or the solver has run for the maximum number of enumerations.

Experimental Setup and Results

To test the performance of our program iterator, the complete methodology is implemented in HerbSearch.jl[1]. This is part of Herb.jl[2] which is a program synthesis library written in Julia. HerbSearch.jl includes a substantial number of search procedure implementations for the program synthesis framework. The most relevant for our purposes is the `BFSIterator`. It is the most basic iterator and therefore a good baseline to compare against. It provides abstract syntax trees from a problem's grammar in increasing order of size. In our experiment, we also employ it as a subiterator for our `GrammarSplittingIterator` during the exploration and exploitation phases.

In addition to search implementations, HerbBenchmarks.jl[3] provides a collection of benchmarks for testing the two iterators. We utilize the PBE SLIA Track 2019 from the SyGuS (Syntax-Guided Synthesis) competition, which includes 100 string-manipulation problems, each with a set of examples and a context-free grammar.

Start → *String*
String → *x* | *""* | *" "* | *"="* | *"-"*
String → *concat(String, String)*
String → *replace(String, String, String)*
String → *at(String, Int)*
String → *toString(Int)*
String → *if Bool then String else String*
String → *substring(String, Int, Int)*
Int → *1* | *0* | *-1*
Int → *Int + Int*
Int → *Int - Int*
Int → *length(String)*
Int → *toInteger(String)*
Int → *if Bool then Int else Int*
Int → *indexOf(String, String, Int)*
Bool → *true* | *false*
Bool → *Int = Int*
Bool → *prefixOf(String, String)*
Bool → *suffixOf(String, String)*
Bool → *contains(String, String)*

Figure 5: String-manipulation grammar from PBE SLIA Track 2019

The 100 grammars from these benchmarks have 27 rules on average and each grammar consistently splits into either 8 or 9 subgrammars using our methodology. Figure 5 depicts one of these benchmarks, with its dependency graph shown in figure 7. One of its subgrammars, highlighted in blue in this graph, is displayed in Figure 6.

When running a problem on one of the iterators, a single example is left out to verify that the synthesized program is not overfitting the given examples. If the synthesized program does not solve this left-out example, the problem is not considered solved.

During the exploration phase, it is not uncommon for a subgrammar to fail to tackle even a single example. Therefore, the user can provide a set of metric functions for vari-

[1]https://github.com/Herb-AI/HerbSearch.jl

[2]https://herb-ai.github.io/

[3]https://github.com/Herb-AI/HerbBenchmarks.jl

Start → *String*
String → *x* | "" | " " | "=" | "-"
String → *if Bool then String else String*
Bool → *prefixOf(String, String)*
Bool → *suffixOf(String, String)*
Bool → *contains(String, String)*

Figure 6: Subgrammar splitted from string-manipulation grammar

ous example output types to assign partial points based on the proximity of a program value to the example output, allowing to more quickly estimate a grammar's relevance. Since 76% of the problems have examples with string outputs, our iterator will use an edit distance metric to compare example output with the synthesized program's evaluation result. Given that the strings being compared might vary in length, we use the Levenshtein distance [4], which is symmetric and ranges from 0 to $\max(|x|, |y|)$. This distance is mapped to a range from 0 to 1, where a score of 0 indicates completely different strings and a score of 1 indicates identical strings.

For the 13 problems with integer outputs, we use the the function $compare(x, y) = \frac{1}{1+|x-y|}$ to give more partial points the closer the return values are to the outputs from the examples. The last 11 problems all of type Boolean will not be given partial points.

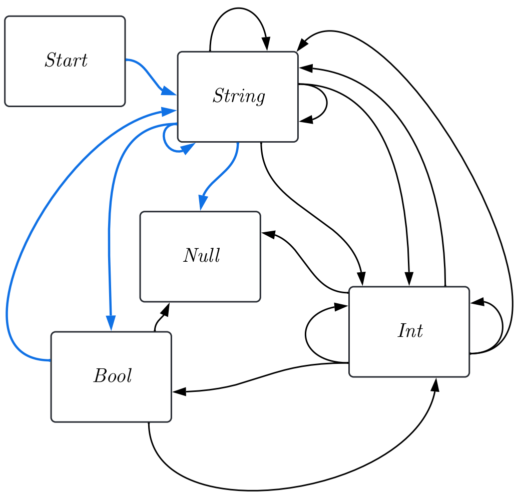

Figure 7: Dependency graph of string-manipulation grammar with one of its subgrammars highlighted in blue

Both iterators are allowed a maximum of 10 million program enumerations per problem. We chose $n = 3$, because the `BFSIterator` rarely uses rules from more than three different subgrammars generated by our `GrammarSplittingIterator`. This results in either $\binom{8}{3} = 56$ or $\binom{9}{3} = 84$ different combinations of three subgrammars that must be explored and exploited, still allowing a significant number of enumerations per subgrammar. Given that scores during the evaluation phase seem to converge quickly and our exploitation time is very valuable, only 5% of the enumerations are used for exploration and the remaining 95% for exploitation. There is no definitive right or wrong when it comes to choosing these values. These values may not be optimal either but testing them across small problem sets indicates they perform best among the considered options. The optimal combination of values also depends on the specific situation, so they are parameters for the user.

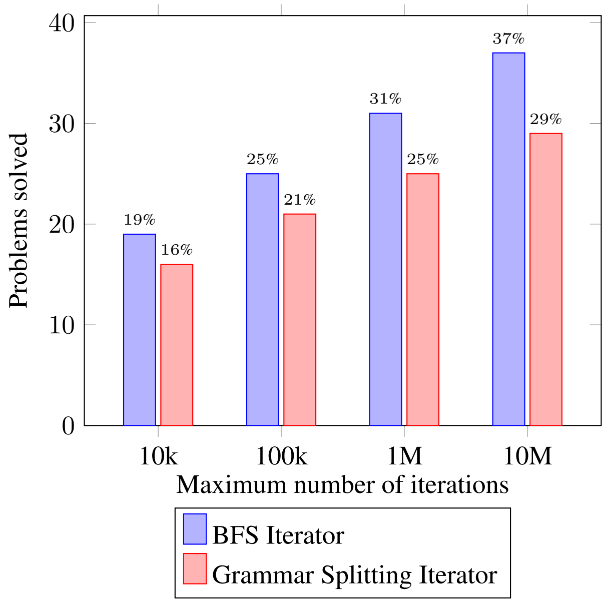

Figure 8: Performance on PBE SLIA Track 2019

Figure 8 shows the percentage of examples solved by both iterators for different amounts of iterations allocated per problem. With the highest experimented amount of 10 million iterations, the `BFSIterator` was able to solve 37 problems, whereas the `GrammarSplittingIterator` only solved 29 problems. For all 28 problems that were solved by both iterators, the two solutions have the same program size but the solutions from `GrammarSplittingIterator` needed on average 7.3 times more iterations to be generated. The average size of the grammars that solved these 28 problems with the `GrammarSplittingIterator` were on average of size 15. Finally, it took the Raspberry Pi 5[4] on which the experiment was run only a little over a second to split all 100 context-free grammars.

Discussion

There were 19 cases where the problem was already solved in the exploration phase. Only two of these solutions required fewer iterations with the `GrammarSplittingIterator` than with the `BFSIterator`. This is presumably because our iterator must check multiple grammars and most likely does

[4]https://www.raspberrypi.com/products/raspberry-pi-5/

not check the one containing the solution first. The other 8 problems that were mutually solved, were solved in the exploitation phase by the `GrammarSplittingIterator`. In one instance, our iterator was particularly fortunate and selected the correct subgrammar first, generating the same solution as the `BFSIterator` in just 808,145 iterations compared to 2,072,958 iterations. It appears that all eight problems would have been solved faster if the correct grammar had been chosen first, taking on average less than a third of the iterations needed by the `BFSIterator`. This is why running the subgrammars in parallel in both the exploration and exploitation could be beneficial for our iterator.

When it comes to program size, it is not immediately obvious why our iterator does not generate solutions that have a bigger program size. Because larger grammars often allow for smaller programs, while we are working with smaller grammars. Therefore, this might not be the case when running on another set of benchmarks.

Responsible Research

It is important for the integrity of the research that the experiment is reproducible. Although, in this case it was executed on a Raspberry Pi 5, it should not matter on which device it is run as we only looked at the number of iterations that the iterators took and not the actual speed in seconds. The iterators themselves, including the benchmarks, are all open source and can be run with the exact same settings as we descriped in the experimental setup section. The Levenshtein distance implementation used in the exploration phases of the experiment is from StringDistances.jl[5].

Conclusion

First of all, an algorithm to split context-free grammars was successfully developped. It makes use of a dependency graph to group rules that can be combined to create programs. Secondly, different sets of combinations of the subgrammars generated by splitting were tried after one another to find a solution from one of them that would take less iterations than by enumerating from the original grammar. Grammars were allocated enumerations based on their performance during exploration. Unfortunately, our own implemented iterator was performing slightly worse than the `BFSIterator` and all these efforts were not yet enough to make the synthesis of programs for our benchmarks more efficient by reducing iterations.

Future Work

Although the worse performance, enumerating from the subgrammars in parallel could potentially allow us to find a solution in one of the subgrammars more quickly than BFS. Therefore, parallelizing the iterator would be part of future work that can be done to improve the strategy. On top of that, one could experiment with different merging strategies. For example, let the iterator run on a set of subgrammars for a while and afterwards merge the most successful grammar with all the others and repeat this step with now a smaller set with slightly more powerful grammars. However, when we do only slight additions to the subgrammars, most of the programs generated by it were also already composable from the previous grammar. Therefore, also a grammar constraint must be implemented that forces a program to contain at least a unique rule from both grammars that are merged.

[5]https://github.com/matthieugomez/StringDistances.jl/

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.