Improving Enumerative Program Synthesis Performance by Extending Grammar from Solutions to Simpler Synthesis Problems

How can such approach be implemented in a synthesis system that cannot benefit from in-advance refinement of the synthesis algorithm parameters

Mert Bora İnevi [1]

Supervisor(s): Sebastijan Dumančić[1], **Reuben Gardos Reid**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A 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, 2024

Name of the student: Mert Bora İnevi
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Reuben Gardos Reid, Soham Chakraborty

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Program synthesis is an important problem in computer science. One method often employed is enumerative program synthesis, which produces a sequence of programs in the target language until one solves the required input-output examples. This can yield undesirable runtimes for some problems that require complex programs to solve them. This research is about how simpler problem solutions can be used to build a library of useful helper functions and to use it for further synthesis in order to reduce the depth of the enumerative search for more complex problems. Drawing inspiration from the work by Ellis et al on DreamCoder, this research describes ways of obtaining simpler programs, and extending the grammar using parts of the solutions. Experiments have been performed on the proposed synthesis algorithm, which is implemented using the Herb.jl framework, and results for each part of the synthesis algorithm being replaced by different strategies are provided. Allowing holes in the new grammar substitutions and using smaller size or frequency as their utility has proven superior. There needs to be more work done about the conciseness of the produced programs.

Introduction

Program synthesis is the process of producing a computer program that satisfies a given list of requirements provided in non-program form. It is considered the holy grail of computer science since it helps automate the creation of computer programs [3]. Current methods include using large language models or evaluating and comparing different programs strategically generated based on the programming language's grammar. This research project involves the latter, analysing each step of program composition including the choice of code snippets, how they are combined, and how it is tested that the resulting program satisfies all the requirements.

Program synthesis within the context of this research requires iteration over the possible programs. Thus, having a good strategy to make the process fast and resulting programs concise and of high quality is important.

The Herb.jl framework, maintained by the supervisors of this research, is one such project where the developers are looking for good approaches to grammar-based program generation. This research aims to collaborate with them in developing and testing an algorithm that can learn common parts between the programs it generates and introduces them into further synthesis in order to improve efficiency. This was inspired by the work of Ellis, et al [2], which estimates probabilities for different symbols based on solutions to randomly generated problems, where the probabilities can be used to build a grammar extension out of common parts of solutions to random problems.

The research tries to answer the following questions, or improve upon the existing literature about them:

- How to pick useful components out of a program?
- How to evaluate the equivalence and similarity of different program snippets and detect replaceable parts?
- How to use common subprograms in further synthesis in order to improve the performance of enumerative program synthesis.

In section 2, there is a discussion of each step a program synthesis approach involves, and considerations to take when implementing the synthesis algorithm with possibility of extending the grammar. Sections 3 includes the description of the experimental setup and extensive testing data for different strategies of program synthesis. Section 4 discusses the implications of the results. Section 5 elaborates on the ethical implications and the reproducibility of this research. Sections 6 reflects on how the new approach compares in performance and resulting code quality to the one by Ellis et al.

Background and Methodology

Program synthesis involves a few steps which can be optimised separately. Roughly, these are the receiving of the specification, processing it in order to feed it to the synthesizer, synthesizing a program, evaluating the program against the specification and applying positive or negative feedback to the synthesizer appropriately, converting the synthesised program to an usable format as necessary, and finally outputting it.

In the context of enumerative program synthesis, the negative feedback constitutes keeping producing new programs that are the valid in the programming language until a correct implementation is found [3, p. 58]. Each program is composed from small program parts. The parts available to produce a program in a given programming language come in the form of a context-sensitive grammar. A context-sensitive grammar includes a start symbol, and consists of mappings which can be used to substitute a symbol with a more concrete code segment. It should also include terminating symbols for which the meaning is implicitly known so the code interpreter can directly evaluate them to a value and/or state change. A top-down program enumerator, which is used for this research, will simply start from the start symbol, and replace it according to the suitable mappings in order to produce new programs, continuing recursively as new substitution symbols are added to the current program [3]. Different strategies can be followed to substitute and order the resulting programs for evaluation, however, all methods are common in that each substitution can add only a fixed amount of code, and more substitutions are necessary to produce more complex programs.

A way to produce more code with fewer substitutions is to have larger, more problem domain-specific substitutions. Automated grammar extension aims to find such larger substitutions that are composed of the existing ones, and add them to the grammar. This is akin to library functions found in a domain-specific program-

ming language which abstract low-level steps needed to perform common tasks, and allows the reuse of functionality with less cognitive overload i. e. less enumeration steps during enumerative synthesis.

For this research it is proposed that extending the grammar before top-down program synthesis can improve the search performance by reducing the number of enumeration steps to reach a program of specific complexity.

Obtaining Simpler Problems for Training

The grammar extension is based on solutions to a set of problems, which can be obtained using various methods like taking them as input or splitting the input problem specification into smaller parts. Such methods of enhancing program synthesis using subsets of the problem domain have been proven successful by various research such as with the work of Shrivastava, et al and Ellis, et al [5][2].

The user can be asked for simpler problems similar to the input problem to extend the grammar with. Even if it may be challenging to apply to the real world solely because of the lack of extra problems in many situations, there is plenty of testing data in the form of benchmarks made of a set of similar problems. DreamCoder, for example, dreams of new problems by randomly creating programs using the posterior probabilities of grammar substitutions, and runs them to form their output in the input output examples [2]. This, and obtaining different problems from the benchmarks, can provide the grammar extension algorithm with novel problems with little risk of overfitting to the original problem and enable finding more versatile subprograms that can be used to solve a wider range of problems.

Another problem is how to solve the simpler problems. A common convention is to use a program enumerator to iterate over all possible programs in the search space, evaluating them on the input, and returning the best performing one. It might sound like an inefficient way of solving the problem, however it works well if the number of tested programs are small (small search space) and the number of examples are small (the odds of the search returning a perfect result early on are higher), which is the motivation behind starting with solving simpler problems which will not require complex solutions. Furthermore, a time limit or number of enumerations limit can be applied while solving the simpler problems, since a good enough solution can still provide useful programs to learn from and exact solutions are not necessary as they do not face the user.

Grammar Extension Selection

DreamCoder utilises an algorithm which finds the probability for each grammar substitution, attempts to solve a set of input problems with timeout, and then adjusts the probabilities accordingly and finds common subprograms to extend the grammar with [2]. The resulting grammar is highly specialized to the domain of input problems, and the training process for each problem domain can take days. This causes a problem since the synthesis program this research is aiming to create should work for synthesising programs for various domains and programming languages. Thus, this research considers a more naive algorithm, as described below.

After solving each simple problem using naive enumerative search with timeout, the proposed algorithm looks for equivalent parts between them. The most direct approach to do this is to compare each subtree of the syntax tree of two programs, and testing if they exactly match. For problems very similar in nature, this can yield common parts at least a few nodes in size, for instance something like the isEven function if the original language doesn't have a built-in function for it. Consider the following two solutions to two hypothetical problems, for example. They both involve checking if some number is even.

```
if length(input) mod 2 == 0 then
  "even length"
else
  "odd length"

if (input * 7) mod 2 == 0 then
  "even"
else
  "odd"

--> no common part is found.
```

This is the case because the two programs do not share any common part which are equal down to the terminating symbols. In these cases, it may be desirable to support finding subprograms with holes in them, meaning the part will be left as a symbol for which substitutions exist in the input grammar, so the synthesizer can create new grammar rules where these holes are filled by the required replacement symbol during enumeration. The common subprogram `boolean = int % 2 == 0` can be extracted from the two programs above, for example, where `boolean` and `int` would be a symbol with available substitutions in the input grammar. The outermost rule used to construct this code segment is a substitution of symbol `boolean` so the same is used for the new grammar substitution rule. The proposed grammar finds such extensions from a pair of programs by iterating over each syntax subtree of two programs and recurses from the roots of the two subtrees, checking if two nodes are exactly the same at current level or the next level is a substitution to the same symbol in the input grammar. In the latter case, the common part is yielded with the different part replaced by the common symbol.

Lastly, the strategy to choose extensions can vary. Likely, there will be a lot of subprograms common between at least two solutions, the number increasing with the number of solved simpler problems. A utility function can be used to rank possible extensions and only pick the top few of them. For example, DreamCoder

uses the posterior probability of each subprogram's constituents, along with the subprogram's size in order to rank them [2]. Based on this, a utility function that maximizes or minimizes either the frequency of, the size of, or the number of holes in the chosen subprograms is possible.

Experimental Setup and Results

A few implementations of a program synthesizer that use the grammar extension method have been tested, and their performance have been compared among each other and to the naive implementation which just uses regular enumerative synthesis on the main problem specification. In figure 1, one can see the synthesizer program structure and different replacement options for each part. To make the measurements scientific, performance comparisons will be based on the iteration counts of loops in the synthesis program instead of wall-clock runtime or similar.

Source code and Usage Summary to Aid Reproduction of Results

The algorithm had been implemented using the Herb.jl framework and made into a Julia package[1]. The package contains some examples which are meant to be run using the REPL. Each example loads in a test benchmark, and maybe other simpler-to-solve benchmarks, runs the sub-problem solving step, then the common subprogram extraction step, and finally attempts to synthesize either one of the sub-problems again using the extended grammar or tackle the main hard-to-solve program. During running, the program prints out how long and how many enumeration steps each subproblem took, and also for each problem solved after grammar extension. It also prints out each grammar extension rule, and their frequency in the sub-problem solutions.

Benchmarks and problems from the 2018 and 2019 SyGuS (Syntax Guided Synthesis) competition have been used for benchmarking. [4].

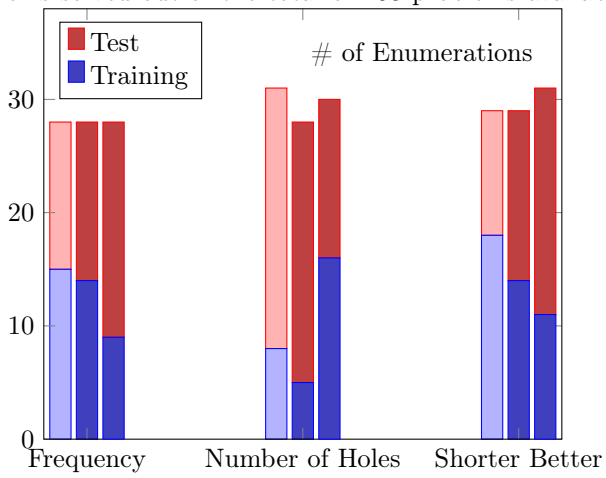
Effect of Allowing or Disallowing Holes in the New Substitutions on the Quality of Extensions

For the testing of extensions with and without holes, a common grammar to solve 8 of the SyGuS 2019 string manipulation problems has been extended on these problems, and the extended grammar was used to solve the same problems. The results can be seen in table 1. With holes being allowed, all the problems seem to require fewer enumerations after the grammar extension. Meanwhile, the number of enumeration steps has increased for most problems when holes are not allowed in the extensions.

The case without holes especially shows that if the extension finder algorithm fails to find generally useful extension, it will just increase the number of enumerations needed for problems which can't make use of such abstractions. Also, considering replaceable parts of subprograms, which are replaced by "holes" when adding to the grammar, proves to be beneficial in providing more opportunities for function library building.

Effect of Utility Function on the Quality of Extensions

For the tests with utility functions, the problems from the bitvector (BV) track of the 2018 SyGuS competition have been used. The problems have been split randomly into a training and test set of equal sizes, and after extending the grammar using the training set with a maximum of 100 000 breadth-first (BFS) program enumerations, it was used to solve the problems in the test set with a maximum of 1 000 000 BFS enumerations. In the following figure one can see the number of training and test problems solved out of the total of 468 problems available.



From these graphs, little difference between the performances of the utility functions can be seen.

BFS enumeration with the original grammar alone can solve 32 problems with the input grammar alone using 1 000 000. The hampering of performance after grammar extension can be attributed to that found subprograms are not useful for a lot of the problems, thus they make the enumeration take very long and cause solutions to be missed with a limited number of enumerations.

If a similar experiment is run on the string transformation benchmarks of the SyGuS 2019 competition, after extending the grammar using all 100 problems in 1 000 000, 11 problems are solved when frequency is used, 14 problems are solved when shorter program length is used, and only 5 problems are solved when the number of holes is used as utility, by doing 10 000 000 BFS enumerations for each. Naive BFS enumeration solves 11 problems in 1 000 000 enumerations with the original grammar. This might indicate that the program length is the best utility to use among the three, but again the difference is small.

[1]Source code can be found at https://github.com/boraini/HerbAutomaticAbstraction

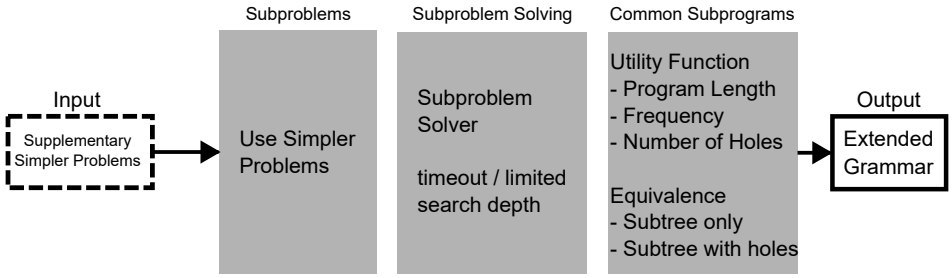


Figure 1: Synthesizer Block Diagram showing Different Options to Test for Each Part

Extension with Holes in Common Subprograms, Frequency as Utility, Use of Provided Simpler Problems

	Before Extension		After Extension	
Problem	Enumerations	Program Length	Enumerations	Program Length
phone 5 short	2475	10	36	6
get first name from name	2699	10	167	6
get first word	2699	10	167	6
remove file extension from filename	2715	10	175	6
34801680	24267	11	647	7
exceljet3	24267	11	647	7
stackoverflow6	24267	11	647	7
28627624 1	45420	12	4515	8

```
Extensions found:
// add some number to the index of some delimiter string
ntInt = indexof(arg1, ntStringDelim, ntInt) + -1
ntInt = indexof(arg1, ntStringDelim, 0) + ntInt
```

Extension without Holes in Common Subprograms, Frequency as Utility, Use of Provided Simpler Problems

	Before Extension		After Extension	
Problem	Enumerations	Program Length	Enumerations	Program Length
phone 5 short	2475	10	5197	9
get first name from name	2699	10	4009	9
get first word	2699	10	4009	9
remove file extension from filename	2715	10	4059	9
34801680	24267	11	817	7
exceljet3	24267	11	817	7
stackoverflow6	24267	11	817	7
28627624 1	45420	12	340544	11

```
Extensions found:
// take from three characters after =
ntString = substr(arg1, indexof(arg1, "= ", 0) + 2, len(arg1))
```

Table 1: Extension with and without holes, Frequency as Utility, Use of Provided Simpler Problems

Discussion

From the available results the best approach for the program synthesis method that is the subject of this research can be derived.

For simpler problem obtaining, it is demonstrated that having problems of similar tasks is beneficial (such as taking the rest of a character string after a delimiting character) in finding common subprograms. For some classes of problems, for example for the string transformation problems found in the SyGuS 2019 competition, extending the grammar can provide the enumerator with shortcuts to solve the problems with fewer enumeration steps. However, for some problems like the ones found in SyGuS 2018 competition, it might hamper performance instead.

When finding common subprograms, replaceable parts of the subprograms should be considered in order to find more useful common subprograms i. e. holes should be allowed in the common subprograms used to extend the grammar.

A concerning thing about the produced programs after grammar extension is that they can get very long due to combining long extensions into a subprogram, where a shorter program would do. For example, in the utility function experiment, the solution for the "remove the file extension from file name" problem was

```
substr(arg1, 1, -1 + indexof(arg1, ".", 1))
```

with the input grammar and

```
substr(arg1, 1,
    indexof(substr(arg1, 2,
        indexof(arg1, ".", 0)
    ), ".", 1
))
```

with the grammar extended using frequency as utility. Both programs do the same thing, but the second one is more verbose due to the grammar extension used not being the best choice for the role it takes in the expression. There are possible solutions for this in literature, some of them having been described in the conclusion.

Responsible Research

The experiments have been designed to be reproducible with minimal alterations of the benchmarks, and by having every test's code separately on the code repository. Anyone who runs a standard Julia interpreter should be able to reproduce the results. The only thing that would be hard to reproduce would be the wall-clock runtimes, which depend on the operating system status, processor speed etc.

One limitation is that the top-down enumeration, which is used for enumerative solving of the simpler problems, is a complex algorithm which there are many ways to implement. Some of the results that were obtained might only be able to be reproducible using the implementation found in Herb.jl.

Since the research does not involve any subjects other than the researchers, there should not be any ethical implication of the research to be concerned about. The source for the testing data has been provided, therefore other researchers can judge how reliable the results on this paper are given the sources.

Conclusions and Future Work

Program synthesis using enumeration has its uses in being simple to implement and the problems it produces being exactly evaluated with perfect information of the environment. For small sets of problems, this research has proven that extension of the grammar from solutions to simpler problems can speed up synthesis, but the main input problem needs to be similar to the simpler training problems, otherwise there will only be more enumeration steps needed using the extended grammar compared to the initial grammar. Possibly with further refinement of the algorithm, the synthesis system proposed in this paper can become useful for a wider range of problem domains.

Furthermore, there is future work done regarding keeping the solutions of the extended grammar concise and of high quality. The proposed algorithm only deals with extending the grammar, but afterwards the same breadth-first (BFS) enumeration algorithm is used to synthesize with the new extended grammar. To the BFS algorithm, the sizes of each substitution in the grammar is insignificant, therefore it can consider programs of high complexity to be relatively simpler programs because they require only a few substitutions to reach with the extended grammar. This should not be mitigated by considering programs containing the new substitutions to be as deep as the substitutions out of the original grammar they consist of. It would remove the advantages of using large chunks of useful subprograms early on to reach solutions faster, by causing the BFS algorithm to take the same number of steps as before, before considering the new substitutions. A smarter synthesis method with the extended grammar is necessary to both make use of the extended grammar and to produce concise programs out of it.

There are some methods to achieve this in literature, and in the future the people behind this research are planning to implement one into the produced grammar-extending synthesis system. A refactoring system similar to the equivalence graph used in DreamCoder can be used to increase the number of common subprograms found between simpler program solutions [2]. This involves the use of equivalence graphs, as described by Detlefs et al [1, p. 58], to keep track of different refactorings of a program that do the same thing, and utilize it to produce simple problem solutions that use some canonical version of each of its parts which are concise in themselves.

Finally, the proposed algorithm could also be made into a program iterator which includes some sort of self-refinement, thus it can accept multiple problems in sequence while also attempting to solve them, forming an interactive system that learns better grammar with each set of problems solved.

References

[1] David Detlefs, Greg Nelson, and James Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52, 09 2003.

[2] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. pages 835–850, 2021.

[3] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[4] Saswat Padhi, Udupa Abhishek, Andi Fu, Elizabeth Polgreen, and Andrew Reynolds. Benchmarks for sygus competition. `https://github.com/SyGuS-Org/benchmarks`, 2019.

[5] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Learning to combine per-example solutions for neural program synthesis. *Advances in Neural Information Processing Systems*, 34:6102–6114, 2021.