Scaling Program Synthesis:

Combining Programs Learned on Subsets of Examples

Tudor Andrei

Supervisor(s): Sebastijan Dumančić, Reuben Gardos Reid

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, 2022

Name of the student: Tudor Andrei
Final project course: CSE3000 Research Project
Thesis comitee: Sebastijan Dumančić, Reuben Gardos Reid, Soham Chakraborty

Abstract

Program synthesis tackles the challenge of generating programs from user specifications, a task proven undecidable due to the exponential search space growth. In program synthesis the Divide and Conquer technique can be employed to prune this search. By decomposing specifications into individual examples, multiple programs are synthesized to solve them separately. Later these small programs are combined into a bigger program which should satisfy all input-output pairs by itself. There have been previous successful attempts at using Divide and Conquer, including combining programs using decision trees. However, a gap persists in the program synthesis community regarding comprehensive frameworks for integrating diverse implementation strategies. This project addresses this gap by incorporating a Divide and Conquer algorithm into a program synthesis library, enabling users to explore different ways of generating the small programs while abstracting the unification procedure. Moreover, we also discuss a "greedy" strategy to generate the programs that will be combined. We found that a Divide and Conquer manages to improve naive search, especially when given more than 10 examples.

1 Introduction

Program synthesis is the field of study of generating programs according to some user-provided specifications. To no surprise, the problem of synthesizing programs is undecidable (Gulwani, Polozov, and Singh 2017) with the amount of possible programs to search increasing exponentially with the amount of lines. Despite this challenge, advancements in computing power, machine learning and novel algorithms have made program synthesis viable and applicable in different areas of computer science such as code completion (Perelman et al. 2012), super-optimization (Massalin 1987) or automating transformations (Gulwani 2011). The idea of producing a program based on the user's intent has even been successfully incorporated into industry standard products such as the FlashFill function in Microsoft Excel (Microsoft 2013).

In this report we will only discuss Programming-by-examples (PBE) synthesis, where the user provides input-output examples and the synthesizer is responsible for finding a program that satisfies all given examples. There are many ways to approach PBE synthesis, but ultimately the problem of a huge search space is still pressing for all methods. To improve the efficiency of searching it is necessary to prune unlikely programs and to explore likely ones. This paper will explore a way to prune the search space by leveraging the *Divide and Conquer* technique. More specifically, the research question being posed is **"How do we combine individual programs learned on each example to form a single program that works on many (all) examples?"**.

In general, *Divide and Conquer* methods are concerned with producing a program that works on small parts of the specification, which are later combined or used as hints for the final program. Recent work (Shrivastava, Larochelle, and Tarlow 2021) in this paradigm achieved impressive results, improving the state-of-the-art models that considered the specification as a whole (such as PCCoder (Zohar and Wolf 2018)). In this paper, the authors assume a specification made up of multiple input-output pairs. They first produce programs that solve individual examples from the specification and then use these programs alongside transformer models to synthesize each line in the final program that satisfies all input-output pairs. There are also disadvantages to this kind of approach. Embeddings have to be learned for the programs, therefore the grammar cannot be changed without retraining the embedding model. Likewise, if the semantics of the programming language change, then the transformer model also has to be retrained. Before this work there have been other successful attempts to learn from examples that do not make use of highly specific machine learning models. In (Alur, Radhakrishna, and Udupa 2017), the authors have managed to combine programs using decision trees. To do so, the grammar must be split into two parts: a term generating grammar and a condition generating grammar. Ultimately, their algorithm arrives at a decision tree where inner nodes are conditions and the leaves are the terms. Each split in the decision tree is equivalent to an `if then else` statement in the final program. This approach has been able to generalize to other grammars and has worked exceptionally well for SLIA (Strings and Linear integer arithmetic).

While both the aforementioned strategies have innovated the field at the time of their release, there are still questions the authors have not expanded upon. A big gap arises from the question of *"How can we generate the starting programs for the Divide and Conquer synthesis?"*. These starting programs will have a great impact on the length of the final program and on the time it takes for the algorithm to finish. The experiments in this paper will investigate how the performance and accuracy of program synthesis are affected when the starting points are programs learned on each example. More background on PBE synthesis is given in Section 2. Section 3 covers the implementation of a divide and conquer method for PBE synthesis, while the experiments and results are discussed in Section 4. Subsequently, Section 5 addresses the topic of responsible research within our study and a conclusion is given in Section 6.

2 Background

This section introduces fundamental concepts required for understanding program synthesis, especially for PBE tasks and syntax guided synthesis. In an effort to standardize program synthesis, there have been conventions established in the field, such as the use of syntax-guided search, SyGuS problems and grammars.

2.1 PBE and syntax-guided synthesis

SyGuS represents a competition for program synthesis. To solve a SyGuS problem the synthesizer is expected to produce a function f that respects the specification of that problem. There are different specifications formats used in SyGuS, however we will expand on PBE specifications as defined by (Alur et al. 2016). For a given problem, a PBE specification ϕ is made up of multiple input-output pairs (referred to as *points*), that tell the synthesizer how the function f should map the inputs to outputs, like in Table 1. The input can be made up of multiple arguments, but the output is

x	f(x)
1	0
2	1
3	1
4	2

Table 1: Example specification for $f(x) = log_2(x)$

```
S ::= T | if (C) then T else T
T ::= 0 | 1 | x | y | T + T
C ::= T ≤ T | C ∧ C | ¬ C
```

Figure 1: LIA Grammar

usually given as a single value. A program is said to satisfy a specification ϕ if $\forall e \in \phi$ the given program solves e. That is, when the program is run on the inputs of e, the output is the same as the output of e. In practice, solvers may try to overfit if given the full specification. For a better evaluation procedure, one example may be hidden in the specification and saved only for evaluation. In SyGuS, programs can get a partial score for a problem, depending on how many examples they manage to fulfill.

Grammars As the name suggests, the syntax guided synthesis approaches rely on some given syntax, more formally presented as grammars. These help the synthesizer by defining the search space (it can only produce programs coming from that grammar) while offering a level of expressivity. A trade-off appears between choosing a larger grammar (more expressivity, but larger branching factor) versus a more compact one (less expresivity, but smaller branching factor). How to choose a good grammar for synthesis is a hard problem and it depends mainly on the application in which the synthesizer is used. SyGuS defines two grammars useful for benchmarks: SLIA and Bit Vector grammars. In Figure 1 a simplified version of the SLIA grammar is shown. Additionally, grammars may contain predefined functions and operators, which need to be interpreted according to some *background theory*. In the case of SyGuS all operators and functions are explained in the standard (Padhi et al. 2023).

Enumerative Solver In the field of program synthesis, a common procedure for solving PBE tasks is the enumerative solver. It is widely considered as the baseline for other novel approaches. The enumerative solver algorithm aims to find an expression from a given grammar that satisfies all examples by enumerating all possible expressions in a certain order. Each expression is preliminarily verified against the points given in the specification; if it fails, it is discarded. If it passes, it undergoes full verification. When the algorithm uses breadth-first order (explores expressions according to their size) then it is guaranteed to terminate and find the smallest solution if one exists. It performs surprisingly well for small to medium-sized problems, but struggles with scalability due to the exponential growth in the number of expressions to check. Algorithm 1 outlines the steps of the enumerative solver on a PBE task.

Algorithm 1: Enumerative solver

```
Input: Grammar $G$, Specification $\phi$
Output: $e$
1:  for $e \in$ Enumerate($G$) do
2:      expt ← false
3:      for $pt \in \phi$ do
4:          if !solves($e$, $pt$) then
5:              expt ← true
6:              break
7:          end if
8:      end for
9:      if expt = false then
10:         return e
11:     end if
12: end for
13: return solution
```

2.2 Divide and Conquer

In Programming by example synthesis, there have been attempts at using Divide and Conquer to prune the search process or aid it by using the answer to smaller problems. In essence, all these methods start from the assumption that it is easier to find a program that works on a single example, compared to finding one that works on the whole specification. If one can do that, then there exist multiple ways to combine the resulting programs to obtain a more general program. One such way is employed in EUSolver (Alur, Radhakrishna, and Udupa 2017), which won the SyGuS competition in 2016 in the PBE track (Alur et al. 2016). A somewhat limiting requirement imposed by EUSolver is that the grammar needs to be partitioned into 2 sub grammars: a *term* grammar and a *predicate* grammar. Expressions that have the boolean type are called *predicates* and expressions that have the same type as the starting variable in the original grammar are called *terms*. The authors do not mention how to perform this split for any grammar, but they claim that for the grammars used in SyGuS it was feasible to do so. Furthermore, their method also assumes that an `if _ then _ else _` expression can be a top-level statement in the grammar i.e. the start variable can be directly expanded into an 'if' statement. With these assumptions in mind, a solver can then enumerate the *term* and *predicate* grammars separately. This significantly reduces the maximum depth, that an enumerative solver would need to reach. The outline of the algorithm is shown in figure 2 taken from (Alur, Radhakrishna, and Udupa 2017). The `while` loop in lines 4 to 5 creates the set of terms. The set of terms has the property that for any 2 terms they do not cover (solve) the same points. The actual combining of terms happens in the `while` loop in lines 6-9. The algorithm enumerates predicates until it is possible to construct a decision tree that classifies each example (point) to a term with perfect accuracy. In the actual SyGuS competition, solvers have access to a verifier which can provide counterexamples for their programs. If the algorithm receives a counterexample it adds it to the set of points and starts the procedure again.

Algorithm 2 DCSolve: The divide-and-conquer enumeration algorithm

```
Require: Conditional expression grammar $G = \langle G_T, G_P \rangle$
Require: Specification $\Phi$
Ensure: Expression $e$ s.t. $e \in [\![G]\!] \land e \models \Phi$
 1: pts ← ∅
 2: while true do
 3:     terms ← ∅; preds ← ∅; cover ← ∅; $DT = \bot$
 4:     while $\bigcup_{t \in \text{terms}}$ cover[$t$] ≠ pts do                ▷ Term solver
 5:         terms ← terms ∪ NextDistinctTerm(pts, terms, cover)
 6:     while $DT = \bot$ do                                        ▷ Unifier
 7:         terms ← terms ∪ NextDistinctTerm(pts, terms, cover)
 8:         preds ← preds ∪ enumerate($G_P$, pts)
 9:         $DT$ ← LearnDT(terms, preds)
10:     $e$ ← expr($DT$); cexpt ← verify($e$, $\Phi$)                 ▷ Verifier
11:     if cexpt = ⊥ then return $e$
12:     pts ← pts ∪ cexpt
13: function NextDistinctTerm(pts, terms, cover)
14:     while True do
15:         $t$ ← enumerate($G_T$, pts); cover[$t$] ← {pt | pt ∈ pts ∧ $t \models \Phi \downarrow$ pt}
16:         if ∀$t'$ ∈ terms : cover[$t$] ≠ cover[$t'$] then return $t$
```

Figure 2: EUSolver algorithm

2.3 Herb.jl

It is worth mentioning the framework that has been central to this project: Herb.jl. Written in the `Julia` programming language, Herb aims to provide a uniform approach to program synthesis. With Herb, a user is able to test their program synthesis algorithms and ideas on different grammars and benchmarks. To achieve this, it provides some useful abstractions and concepts to work with: `Grammars` and `Iterators`.

Grammars. In Herb all synthesizers are syntax-guided and have support for 2 types of grammars: context-free grammars and context-sensitive grammars. Meta programming in Julia makes the process of creating a grammar easy as it is close to how one would write it on paper (Listing 1). For context-sensitive grammars, a user has the option of adding different constraints. Considering the grammar in Listing 1, we could add a constraint which enforces that after expanding the multiplication rule, we must also expand the last rule (`Int = x`). This would only produce programs where if a multiplication is present, then there is also the `x` symbol.

Listing 1: Example grammar in Herb

```
g = HerbGrammar.@cfgrammar begin
    Int = |(0:9)
    Int = Int + Int
    Int = Int * Int
    Int = x
end
```

Iterators. They abstract the process of producing programs from a grammar. Iterators can come in different forms and operate in different ways, however they all must produce programs when requested. There are lots of useful iterators already implemented in Herb. Going back to the enumerative solver, its counterparts in Herb are the DFS and BFS iterators. Consider the BFS iterator: given a grammar this iterator will subsequently produce expressions in the order of their depth (number of rule expansions). For instance, if the grammar in Listing 1 is given, then the iterator will produce expressions: `0 ,1, 2, 3, 4, 5, 6, 7, 8, 9, x, 0+0...`

3 Implementing EUSolver in Herb.jl

This section describes our implementation of the EUSolver in `Herb.jl`, with a couple of modifications. Firstly, EUSolver was designed to work in the SyGuS competition where an intermediate verification step is allowed: a program can be prematurely outputted and the judge will either approve it or provide a counterexample in the form of a *point*. This was not implemented into Herb as there is no judge involved in the solving of a problem. Secondly, we expand on some points that were not fully discussed in the original paper such as how to effectively split a grammar and term generating strategies. Lastly, we offer some insight into a small performance improvement that has been implemented.

3.1 Partitioning the grammar in practice

A requirement of the algorithm is to have separate grammars for *terms* and *predicates*. To implement this we used a particularity of the SyGuS grammars from the PBE track. In Listing 2 an actual grammar from the SyGuS competition

Listing 2: SyGuS grammars

```
Start = ntString
ntString = _arg_1 | "" | " " | "."
ntString = concat(ntString, ntString)
ntString = replace(ntString, ntString, ntString)
ntString = at(ntString, ntInt)
ntString = int_to_str(ntInt)
ntString = ntBool ? ntString : ntString
ntString = substr(ntString, ntInt, ntInt)
ntInt = 1 | 0 | -1
ntInt = ntInt + ntInt
ntInt = ntInt - ntInt
ntInt = len(ntString)
ntInt = str_to_int(ntString)
ntInt = ntBool ? ntInt : ntInt
ntInt = indexof(ntString, ntString, ntInt)
ntBool = true | false
ntBool = ntInt == ntInt
ntBool = prefixof(ntString, ntString)
ntBool = suffixof(ntString, ntString)
ntBool = contains(ntString, ntString)
```

is displayed. To implement the grammar splitting, we can consider two iterators over the same grammar, with a small modification: the iterators have different starting variables. The term iterator will use the `Start` symbol as its first expansion, while the predicate iterator will use the `ntBool` symbol. In the actual implementation the user is expected to provide these two symbols. This allows for great flexibility and it is also simpler to handle programatically as there is no need to actually create 2 grammars. We found that all SyGuS grammars can be 'split' using this procedure, since they support an expression like `c ? a : b` or `if c then a else b` as a top level statement. The symbol that generates expressions of `c` can be used as the starting symbol for the predicate iterator.

3.2 Generating terms

A great deal of experimentation is possible in generating the initial terms. In this paper, a much simpler yet more extensible approach is taken. The implementation does not require the term iterator to work as an enumerator, that repeatedly produces programs without any pruning or searching techniques. Instead, it allows any search procedure in `Herb` to be given as a term iterator. This offers a great amount of flexibility as a user can choose between stochastic iterators, genetic algorithms or a Breadth-First iterator in which case it would work the same as the EUSolver.

Having as few terms as possible is desired as it would lead to a shorter program, less likely to overfit on the examples. To create such a set of terms two approaches are possible. A *greedy* approach would consider new terms that solve at least one different example compared to any other term previously added in the set. This leads to a set of many terms, which will also need more predicates for unification. Alternatively, another approach could keep generating terms that fulfill at least an example. For a term to be added to the set, it is not necessary to solve a different example compared to other added terms. After a fixed amount of enumerations, we can find a minimal subset out of the terms collected so far that solves all examples.

Example Take for instance the synthesis of the `min(x, y)` function. A *greedy* procedure will produce the terms: `0, -1, x`. `0` and `-1` are not desirable, but *greedy* will stop looking for better terms because it already has a set satisfying the whole specification. On the other hand, the subset method will generate a plethora of terms until a certain threshold of iterations is reached. Even though it has to generate `0, -1, x, y, ...` it will also consider the terms `x` and `y`, which are ideal since the best program is: `x < y ? x : y`. In this paper the first *greedy* approach is implemented. The clear advantage of this method is that it requires less enumerations and it is clear when to stop producing new terms. In turn however, the final programs it will produce will generally be larger compared to the minimal subset method.

x	y	min(x, y)
0	1	0
0	-1	-1
3	4	3

Table 2: Specification for `min` function

3.3 Generating predicates

Following the EUSolver paper, it can be concluded that once the initial terms are generated the unification procedure can work as a separate unit. Meaning, the initial programs can be provided by all sorts of iterators/solvers (enumerative, stochastic, genetic, etc.). The only requirement for this procedure is that the user provides the list of examples and a way to generate distinct predicates. The latter can be easily achieved with a simple Breadth-First enumerator over the *predicate* grammar. This is exactly how unification was implemented in Herb: a procedure that can be used by (almost) any iterator as long as the 2 aforementioned requirements are satisfied.

Predicate batching In the original paper the algorithm states that after generating a predicate we should learn a decision tree. This becomes cumbersome in practice, especially in a garbage collected language as `Julia`. A possible optimization is to generate more predicates at once and then run the unification procedure. In practice this has the effect of allowing the predicate generator to produce better conditionals, which will be selected as a split in the decision tree because of their information gain. This small improvement proves very beneficial in cases where the terms are very different from example to example. Therefore, more distinct terms in the final program are needed and therefore more conditions are used to split them.

4 Experimental Setup and Results

In the following section, we will present and discuss the results of the experiments we have performed. Firstly, the metrics and datasets will be explained, then information about the setup will be presented. Finally, a discussion on the results will be made.

4.1 Setup

Along with an implementation of the divide and conquer method, we also present its performance against the enumerative solver on different datasets. It is important to measure this as it reveals the advantages and the drawbacks of this approach.

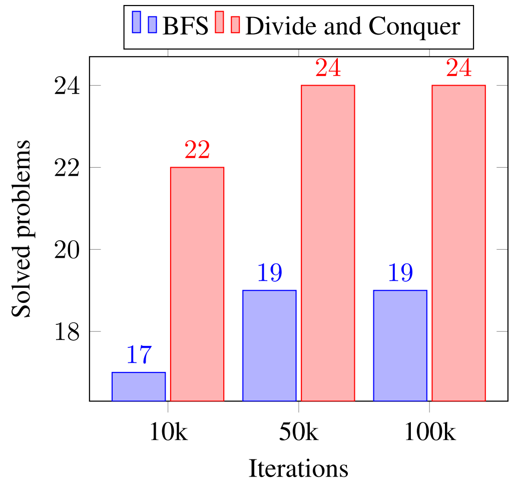
Datasets. The datasets used are “PBE_SLIA_Track_2019” and “PBE_BV_Track_2018”, from HerbBenchmarks. They consist of 100 and 468 problems respectively. The SLIA track consists of medium to hard difficulty problems that require the program to perform logic on 3 types: Integers, Strings and Booleans. The BV[1] track comprises of problems in which the program is required to perform multiple bitwise operations on a single starting integer value to obtain another integer value as the output. A particularity of the latter track was that some problems contained more than 500 examples. Given that it is not realistic for a user to provide that many examples, these problems were filtered out from the dataset reducing the number of problems to 317.

Enumerations. A typical measure of performance for a program synthesizer are the number of enumerations it performs to reach the correct program. For the divide and conquer method the number of enumerations is defined as the sum of the enumerations performed by the term iterator and the predicate iterator. In our divide and conquer method a BFS iterator was used as a term and predicate iterator, so the number of enumerations is directly comparable with a simple BFS iterator. We have tested both methods with different number of iterations. This means that for each problem the iterator is allowed to perform at most a 10k, 50k or 100k enumerations.

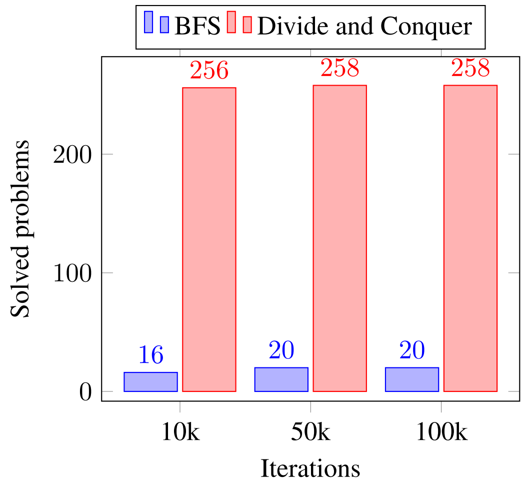
4.2 Evaluation.

To evaluate one of the methods on a problem we cannot simply use the full specification. If we do, it will encourage the

[1]Short for Bit-Vector



(a) SLIA (100 problems)



(b) BV (317 problems)

Figure 3: Problems solved from SyGuS benchmarks according to our evaluations scheme. We set three thresholds for the number of iterations: 10k, 50k and 100k.

solvers to overfit on the given examples. Instead we propose an evaluation scheme similar to the field of machine learning. Assuming a problem has n examples, we give the solvers $\lfloor 0.9 \cdot n \rfloor$ examples (randomly selected) to produce a program. To evaluate their solution we run it on the full specification with n examples. The problem is considered solved if it correctly solves the n examples. In all the experiments conducted below this method of evaluation has been used.

4.3 Results

To evaluate our implementation of the divide and conquer method we formulate two questions: *"What is the solving performance against enumerative solver?"* and *"How good are the solutions produced by the divide and conquer approach?"*.

Performance. To answer the former, in figures 3a and 3b, bar charts of the performance on the 2 datasets are shown. The value of the bars indicate the number of problems each method has solved, while the bottom axis shows the amount of enumerations it was allowed (per problem). The divide and conquer performs slightly better on SLIA track, with the difference being much clearer on Bit Vector track. Furthermore, Figures 4 and 5 show the cumulative number of enumerations both algorithms make. The divide and conquer method takes significantly less iterations over the whole datasets (1M less for SLIA and 27M less for BV).

Quality. To measure the quality of solutions we are comparing the length[2] of the produced programs (from the problems both methods managed to solve). The result is shown in the scatterplots 6a and 6b, which correlate the length of the 2 obtained sizes for a program. For both datasets, we can observe that the divide and conquer method produces larger programs compared to regular BFS, which is expected

[2]The length of a program in this context refers to the number of expanded rules in the grammar including terminal symbols

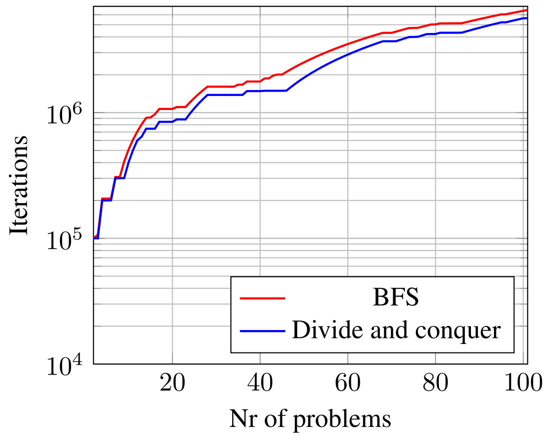


Figure 4: Cummulative iterations on SLIA

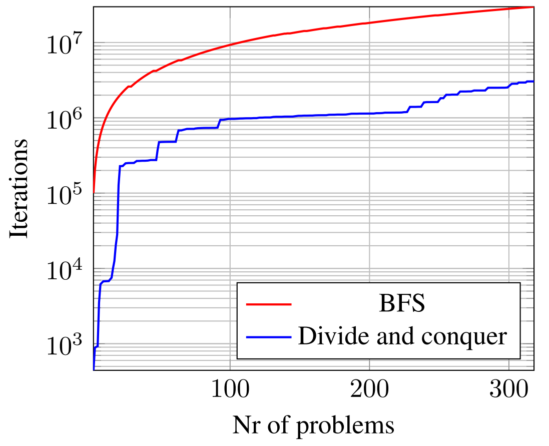
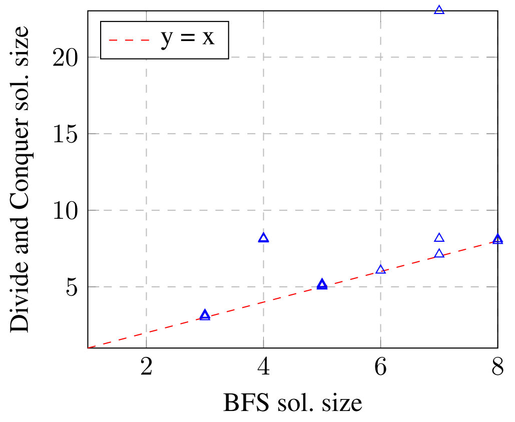
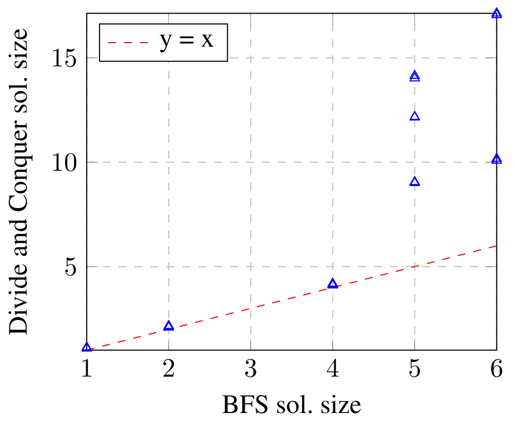


Figure 5: Cumulative iterations on BV



(a) SLIA



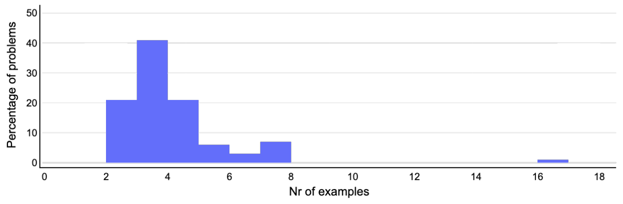
(b) BV

Figure 6: Correlating the solution size of the programs both methods produce. x-axis: BFS, y-axis: divide and conquer. Each point represents a problem both methods managed to solve.

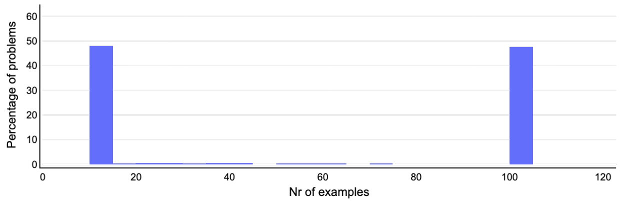
since BFS is guaranteed to obtain the smallest solution. If we closely analyze the scale of the difference in length we can observe that in most cases it is a factor of 2 or 3. On the Bit vector track we clearly see this factor when the size of the smallest program is largest (i.e. length 5 and 6). This can explain how the divide and conquer method manages to outperform BFS on the BV track: divide and conquer is able to construct smaller programs which are successfully combined by 1 or 2 conditions.

4.4 Discussion

Even though the divide and conquer algorithm boasts better results in the performance metrics compared to BFS, it is interesting to look at the big difference over the two datasets. A revealing factor may lie in the number of examples each problem has 7.



(a) SLIA



(b) BV

Figure 7: Distribution of examples across the datasets

SLIA mostly has problems with less than 8 examples, while for the BV dataset the number of examples is at least 10. This big discrepancy might explain the difference in performance: The more examples a problem has the harder it is for BFS to find a program that solves them all. The divide and conquer method can generate the necessary terms in much fewer iterations than BFS can find the optimal program. This behaviour is clearly visible when the problems have more examples. Another metric that points to this conclusion is the solution sizes 6a 6b. On the SLIA track, the divide and conquer method has mostly produced programs the same size as BFS. This means it was quicker to find a program that solves all examples, compared to finding the necessary terms and predicates. On the BV track we see more instances where it was quicker to produce the terms and predicates than finding the complete program. Keeping in mind that the two scatterplots show only problems that both methods managed to solve, we can conclude that a lot of the problems not included were much easier for the divide and conquer to solve compared to BFS.

5 Responsible Research

In our research on program synthesis, we emphasize the importance of responsible research practices. Our experimental methodologies are designed to ensure high reproducibility, particularly when evaluated against the SyGuS benchmarks. This reproducibility is achieved through extensive documentation, publicly available code repositories, and an easy procedure of repeating the experiments. The experiments were ran in the Julia programming language using the `Herb.jl` framework. The Gitlab repository [3] contains all the necessary dependencies, with the exact branch and version being specified. Results were computed on a M2 Mac Studio machine. This should facilitate verification and extension by the research community in program synthesis, such is the aim of the framework `Herb.jl`. Nonetheless, it is important to recognize the inherent limitations of our approach. While our techniques demonstrate good performance on the

[3]The repository with the code can be found here.

SyGuS benchmarks, there exists a substantial likelihood that these methods may not generalize effectively to arbitrary grammars beyond the scope of these benchmarks, especially those that produce stateful programs. This constraint highlights the necessity for ongoing evaluation and adaptation across a diverse array of grammars to enhance the generalizability and practical applicability of the divide and conquer method.

6 Conclusions and Future Work

This paper presented an adaptation of EUSolver in Herb.jl. Starting from the idea that it is easier to find small programs that solve parts of the specification and then combine them, this method was able to scale the enumerative search considered as baseline. We have mentioned the advantages of implementing algorithms into a unified framework for program synthesis, such as being able to easily test the method on different datasets or trying different sub-iterators in the case of the divide and conquer method. From the evaluation results we have concluded that as the number of examples grows, the efficiency of the divide and conquer approach is clearly visible. We found that decoupling the generation of conditions and statements heavily reduces the search depth, a figure which is definitive for the performance of exponential algorithms.

While implementing this algorithm we also found points for further research. Firstly, an automated procedure that determines whether a grammar can be split into a term and predicate grammar (and if so perform the split) would greatly improve the usability of this algorithm, which in turn allows it to be integrated into consumer grade products. Secondly, a great area to be investigated is the generation of terms. A specialized solver that works really well on problems with a single or very few examples has the potential to significantly improve the divide and conquer method. Solvers employing machine learning techniques could prove very useful for this scenario since there isn't a strict requirement to synthesize a general program, but rather one that solves a single example.

References

Alur, R.; Fisman, D.; Singh, R.; and Solar-Lezama, A. 2016. SyGuS-Comp 2016: Results and Analysis. *Electronic Proceedings in Theoretical Computer Science*, 229: 178–202.

Alur, R.; Radhakrishna, A.; and Udupa, A. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In Legay, A.; and Margaria, T., eds., *Tools and Algorithms for the Construction and Analysis of Systems*, 319–336. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-662-54577-5.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, 317–330. Austin, TX, USA.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1-2): 1–119.

Massalin, H. 1987. Superoptimizer - A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, 122–126. Palo Alto, California, USA.

Microsoft. 2013. Using Flash Fill in Excel. https://support.microsoft.com/en-us/office/using-flash-fill-in-excel-3f9bcf1e-db93-4890-94a0-1578341f73f7. Microsoft Support.

Padhi, S.; Polgreen, E.; Raghothaman, M.; Reynolds, A.; and Udupa, A. 2023. The SyGuS Language Standard Version 2.1. arXiv:2312.06001.

Perelman, D.; Gulwani, S.; Ball, T.; and Grossman, D. 2012. Type-Directed Completion of Partial Expressions. In *PLDI'12, June 11-16, 2012, Beijing, China*.

Shrivastava, D.; Larochelle, H.; and Tarlow, D. 2021. Learning to Combine Per-Example Solutions for Neural Program Synthesis. In *Advances in Neural Information Processing Systems*, volume 34, 6102–6114.

Zohar, A.; and Wolf, L. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In Bengio, S.; Wallach, H.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.