Efficient Program Synthesis via Anti-Unification

Enhancing Domain-Specific Language-Based Synthesis by Identifying and Utilizing Common Patterns

Radu Nicolae

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: Radu Nicolae
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Reuben Gardos Reid, Soham Chakraborty

Abstract

Program synthesis is the process of constructing programs that provably satisfy a given high-level user specification. Recent work in this domain has focused on utilizing domain-specific languages to guide the search procedure. This study proposes a novel approach to enhance the efficiency of such search procedures. By utilizing anti-unification, which is the process of generating the least general pattern between two symbolic expressions, this work aims to find common subcomponents to enhance the language used in program synthesis to reduce search depth and improve performance.

Introduction

Program synthesis, often regarded as the "holy grail" of computer science, is the automated process of creating a program in a specific programming language based on user-defined specifications. This process entails the act of searching over the space of all the programs to generate one that is consistent with the constraints derived from the user intent [8].

As explained by Cristina David and Daniel Kroening [5], program synthesis has a wide range of real-world applications. It could have a significant impact in various domains, including:

- **Code Refactoring**: Under the pressure of deadlines, software developers tend to find quick solutions for problems without putting an accent on writing good code. For this reason, program synthesis could be used to refactor existing code to enhance testability, maintainability, and extensibility while retaining the semantics of the program.
- **Digital Controllers**: Due to the difficulty of striking a balance between a digital controller and the physical machine that it operates, control engineers can rely on program synthesis to produce correct implementations of the controllers.
- **Data Manipulation**: Utilizing data manipulation tools is a complex task for non-expert users as they often come with a multitude of features that overcomplicate the process. For this reason, Program synthesis could be used in these tools to automate complex data manipulation tasks.

Many approaches to program synthesis have emerged over the years, ranging from deductive synthesis and transformation-based synthesis, which require a complete formal specification of the problem, to inductive synthesis, which relies on inductive specifications such as input-output examples. Recent approaches also allow the user to provide a skeletal structure (grammar) of the space of possible programs alongside the specification [5].

Among these methods, traditional program synthesis techniques often utilize domain-specific languages (DSLs) to guide the search process. A persistent problem with the guided search method is that it faces challenges when applied to larger codebases due to the extensive search space.

Drawing inspiration from the duplicate code detection algorithm [2], which uses anti-unification to identify common code patterns, this work introduces an algorithm that detects common expression patterns in programs generated by a DSL-based synthesizer and transforms them into new additions to the language. The rationale behind this method is that the newly compressed components will serve as valuable abstractions by enabling the synthesizer to find solutions more efficiently. Since these components are composed of multiple expressions, generating more extensive and complex programs becomes easier. Moreover, the recurrence of these common components across numerous programs indicates a certain 'quality' that can help guide the synthesis process toward better solutions. While the idea of generating a specialized library of code abstractions has been explored in DreamCoder [6] and further refined in LLMT [3], the approach proposed in this study distinguishes itself by being agnostic to the search procedure used to iterate through the program space. While this feature makes the algorithm lack the depth that the other approaches have, it offers the flexibility of being able to be applied over other program synthesizers to improve them.

Background

As mentioned in the previous section, program synthesis is the automated process of finding a program that satisfies the user specifications. This paper's approach will be concerned with program-by-example and syntax-guided synthesis.

Syntax Guided Synthesis (SyGuS) [1] is a framework designed to tackle the program synthesis problem by combining a semantic correctness specification with a syntactic template. In SyGuS, the user provides several specifications representing the desired program's behavior and a grammar that defines the space of possible implementations. This allows the synthesis process to be guided by specifications and structural constraints, significantly narrowing the search space.

There are multiple ways to represent the user specification in the SyGuS framework. However, we will expand on Program by Example (PBE) [7]. In PBE, the user specifies the desired program behavior through input-output examples rather than through explicit logical formulas or detailed specifications. This approach makes program synthesis more accessible to users who may not have expertise in formal methods or programming. It also enables the synthesizer to quantify a program's quality based on the number of examples it solves.

Given a grammar representing the Domain-Specific Language (DSL) and a set of input-output examples, the synthesizer will then use an efficient algorithm to search through the program space to find a program that correctly satisfies all the provided input-output examples. By utilizing the structural constraints imposed by the grammar and the behavioral constraints from the examples, the synthesizer will have a robust framework that will allow it to generate and test programs efficiently.

Anti-unification is the process of generating a new symbolic expression from a set of other symbolic expressions that maintains certain commonalities with all of them [4]. This process was first introduced by Plotkin [10] and Raynolds [11].

The anti-unification approach's success is due to its compression. By generating the least general patterns between

multiple programs, we create new functions for the program synthesizer to use. However, too many overly specific functions in the DSL could slow the synthesizing process. Furthermore, quantifying the "quality" of a function in the context of finding a correct solution for a user-defined problem is complex, making the process of applying anti-unification a challenging task.

One of the state-of-the-art approaches in the field of program synthesis that implements a DSL-based search is DreamCoder [6]. The algorithm is an iterative program synthesizer that takes as input a corpus of problems, each specified by a set of examples (user specifications), and outputs a library of program components and a neural search policy that can efficiently solve other similar synthesis problems. It comprises a latent variable generative model, and a neural network called a recognition network. Each cycle of the algorithm starts with the wake phase, and it utilizes the DSL and the recognition network to solve as many of the problems provided as input. It then ends with the sleep phase, which consists of two distinct steps. First is the abstraction sleep step, which grows the DSL utilizing a refactoring algorithm based on E-graph matching. Second is the dream sleep step, which trains the neural network to improve the system's synthesis skills.

One study that leveraged anti-unification with great success in the field of program synthesis is *BABBLE: Learning Better Abstractions with E-Graphs* [3], outperforming DreamCoder. The work introduces an enhanced version of the abstraction sleep step of DreamCoder named library learning modulo theory (LLMT). This new library learning algorithm additionally takes a domain-specific equational theory as input and utilizes it to create e-graphs that encapsulate all equivalent terms. It later applies anti-unification on the generated e-graphs to propose new candidates for the learned library. It chooses between them based on the reduction that the functions provide to the corpus of problems.

Drawing inspiration from the work mentioned above and other studies that applied anti-unification in other domains [2], this paper aims to propose a novel approach that optimizes the process of program synthesis while being agnostic to the search procedure.

Problem Description

To effectively apply anti-unification to optimize the process of synthesizing, three main challenges should be addressed.

First, *which programs should be considered for anti-unification?* To answer this question, we first need to define what is a "good" program. Since we cannot estimate how close a program is to the correct solution, we can only check which user specifications it satisfies. Having at least one of the specifications satisfied is an acceptable condition for it to be used for the anti-unification process. However, another factor that should be checked is the number of programs collected. Two approaches are possible. Either collect an arbitrary amount of programs or until the union of all the specifications the programs solve is equal to the set of all user specifications.

Second, *On which criteria should a common pattern be accepted into the DSL?* As mentioned in the previous section, introducing too many overly specific functions into the DSL could prove detrimental and slow down the synthesis process. Two metrics that have seen use in other works that leveraged anti-unification are compression (Dreamcoder [6] and Babble [3]) and the number of substitutions (Duplicate code detection [2]). Compression is the number of nodes that would be reduced in the program's Abstract Syntax Tree (AST) if the common pattern is exchanged with its corresponding function produced by anti-unification. The number of substitutions could be seen as the number of arguments that the function generated by anti-unification. Further details about what is substitution in the domain of anti-unification can be found in [2] [4]. This metric ensures that the resulting functions are human-interpretable and do not become too general.

Third, *How often should a pattern appear to be common?* Finding a common pattern in a sizable set of collected programs could be complex. Since the programs are not collected based on their structure, they could vary a lot, resulting in reduced chances of finding a common pattern in all of them. To tackle this problem, common patterns could be found in subsets of arbitrary sizes. This approach is inspired by Babble [3] in which patterns found between any two programs are considered to be added in the component library.

Anti-unification Meta-iterator

In this section, we will present the algorithm that leverages anti-unification to speed up the process of synthesizing programs.

Utilizing the terminology used in Herb.jl[1], the framework in which this method is implemented, an iterator is the main structure that defines the search procedure of a synthesizer. This study aims to create a "meta-iterator" in the sense that anti-unification will not be the one that guides the search procedure. Instead, it tries to improve another iterator by sequentially improving the DSL it utilizes to reduce the depth of the search procedure.

At each step of the algorithm, at least one new contribution is inserted into the DSL. The process of finding these contributions has three sections.

Collecting programs

The algorithm will use the iterator provided to generate problems and evaluate them based on user specifications. If at least one of the specifications is satisfied, the program and a set of all of the specifications that it satisfies are kept in a list for further processing.

If the union of all specifications solved by the collected programs is equal to the list of specifications provided by the user or a certain number of programs have been stored, the procedure stops. If a program that solves all of the user specifications is found, the procedure stops.

Applying Anti-unification

After having a list of programs, we can apply anti-unification to subsets of the collected programs to find func-

[1]Herb.jl, "Herb.jl Documentation," Herb.jl: A Unified and Universal Framework for Program Synthesis, https://herb-ai.github.io.

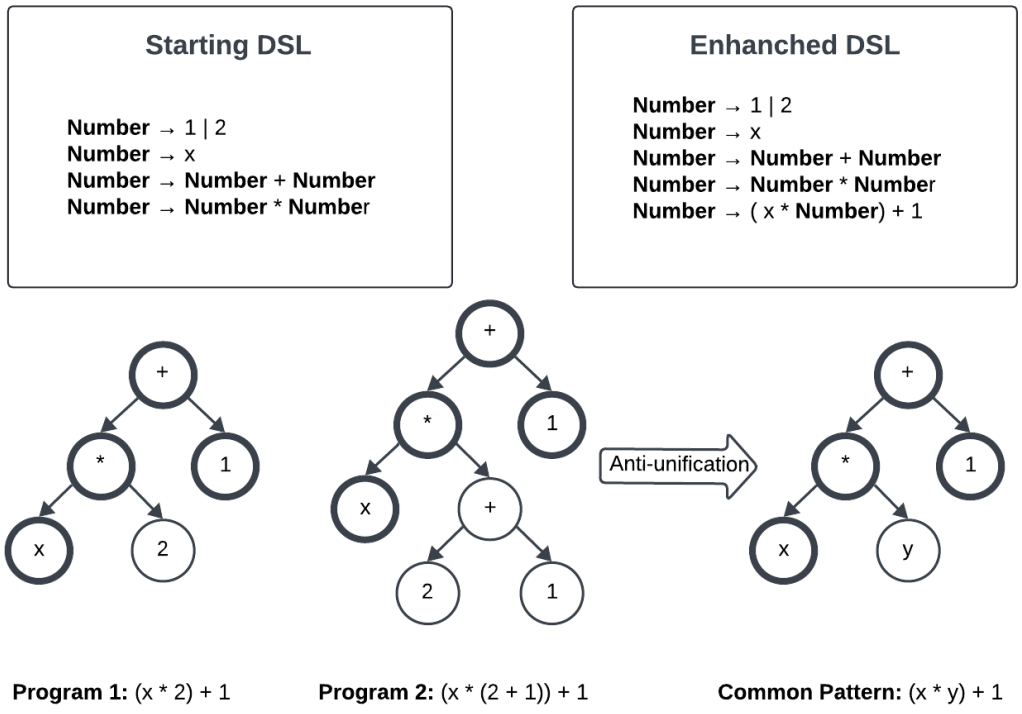

Figure 1: Example steps in the process of enhancing the DSL of a synthesizer using anti-unification

tional common patterns. By establishing the size of the subsets, every combination of programs will be represented as an abstract syntax tree (AST) and then undergo anti-unification.

The process of finding common patterns between multiple programs comprises the following components.

- The anti-unification algorithm is inspired by MST [2]. It takes as input two ASTs, and starting from the roots, it checks pairs of nodes for equality. In the context of an AST, two nodes are similar if they represent the same function and have an equal number of descendants. If these conditions are met, the function will become a part of the common pattern. Furthermore, pairs of their child nodes will be made based on their ordering and checked for the same conditions. A variable node will be added to the common pattern if the nodes checked for equality differ.
- To collect all the common patterns between two ASTs, the anti-unification algorithm will be applied between all the pairs of subtrees that do not belong to the same tree. The pattern must be checked to satisfy two user-defined thresholds to be considered for further processing. First, the number of non-variable nodes from which they are composed should be greater than a particular value. Second, the number of variable nodes they contain should be smaller than a specific value.
- To enable the algorithm to find common patterns between more than two ASTs, the proposed solution takes an iterative approach. For each AST that does not belong to the initial pair, we will apply anti-unification between itself and all of the common patterns found. Following this logic, the resulting patterns will contain commonalities with all of the ASTs that have been iterated through.

Enhancing the DSL

After applying anti-unification, a list of candidate common patterns will be composed. A set of all the user specifications its parent programs satisfy will be kept for every common pattern added to the DSL. A candidate pattern will be discarded if the set of user specifications that its parent programs satisfy is already a part of the abovementioned list. The reasoning behind this choice is first to limit the number of DSL additions. More importantly, even if it limits the potential of finding better patterns for that specific set of specifications, it also limits the addition of equivalent patterns that contain redundant steps. This condition is essential as once a pattern has been added to the DSL, the process of finding equivalent programs to its parent programs will be much faster, leading to an overflow of additions to the DSL that will not enhance its efficiency but, instead, slow down the process of synthesis.

If no candidate patterns are found, the current list of collected programs will be discarded, and the algorithm will restart from the first section. Once at least one pattern is found, they will be restructured into new functions to be added to the DSL. The root of the AST of the common pattern will dictate the function type, while any variable node

in its structure will become the function's argument.

Experimental Setup and Results

The experimental segment of this study aims to assess the proposed anti-unification meta-iterator method for improving DSL-based program synthesizers. The primary objective is to highlight its strengths and weaknesess by evaluating how effective it is compared to the stand-alone iterator it utilizes.

The primary metric on which the programs will be compared will be the number of enumerated programs it takes until a solution is found or a limit on the number of enumerations has been achieved. The comparison will be made on the problems proposed in the SyGuS SLIA and BV track benchmark[9]. At the same time, the iterators used for the assessment will be BFSIterator and DFSIterator provided by the Herb.jl framework.

BFSIterator is a top-down iterator that, given a grammar (DSL) and a starting symbol, searches the program space in a breadth-first search style, returning programs in increasing order of size. Similarly, the DFSIterator is a top-down iterator that also takes a grammar and a starting symbol as input. However, it searches the program space in a depth-first search manner, prioritizing exploring the largest program. In the SyGuS SLIA track, there are 100 problems related to string operation, while in the BV track, there are 753 problems associated with Bit-manipulation. However, due to their complexity, we will use only the first 250 problems for our comparison. Each issue is defined by a set of input/output examples and has a specific grammar attached to it.

The setup used for the Anti-unification meta-iterator algorithm will be running with a minimum 4 non-variable nodes in the AST of the common patterns, the maximum number of collected programs will be 3, and anti-unification will be applied on subsets of 2 programs at a time with a maximum of 5000000 enumerations. The sub-iterators will be running with default parameters, with the same limitation of a maximum of 5000000 enumerations.

BFSIterator Comparison on SyGuS SLIA track

Out of the 100 problems, the Anti-unification Meta-iterator solved 49 problems, while the BFSIterator solved 43 problems. Excluding the problems for which the anti-unification did not find any common pattern, there are 37 problems. Furthermore, we can prune the set of problems to eliminate the problems that both the iterators did not solve. In the end, the final subset of problems we will look at is made up of 21 problems. The subset comprises seven problems solved only by the Anti-unification Meta-iterator, one problem solved only by the BFSIterator, and 13 problems solved by both.

A problem highlighting a bottleneck of the algorithm proposed in this paper is called 'problem_count_line_breaks_in_cell' and is the first problem in Figure 2. It is the only one solved by the BFSIterator and not by the anti-unification meta-iterator. This result is caused by one of the examples from which the problem

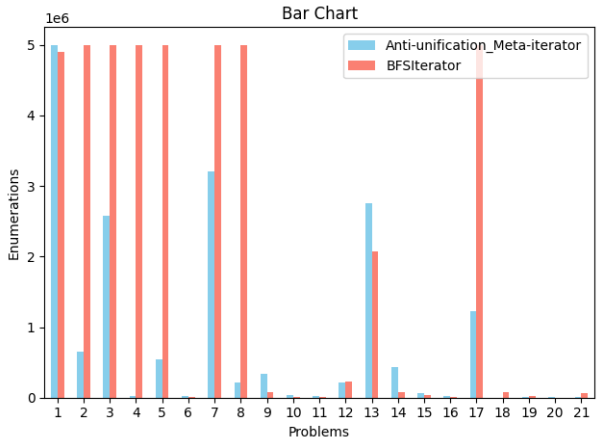

Figure 2: Enumeration Counts for BFSIterator vs. Anti-unification Meta-iterator on SyGuS SLIA Track

is made. The answer to it is simple, leading the iterator to collect a lot of redundant programs that unintentionally solve the example. The single addition to the DSL found is: 'if false then str_to_int_cvc(ntString) else 1' which will constantly evaluate to 1, independent of the values that ntString takes. Here, ntString is a placeholder that the program synthesizer will try to replace with other elements in the grammar that evaluate to a string. Instead of being beneficial, this pattern will have an adverse effect, increasing the number of enumerations it takes to find a correct solution.

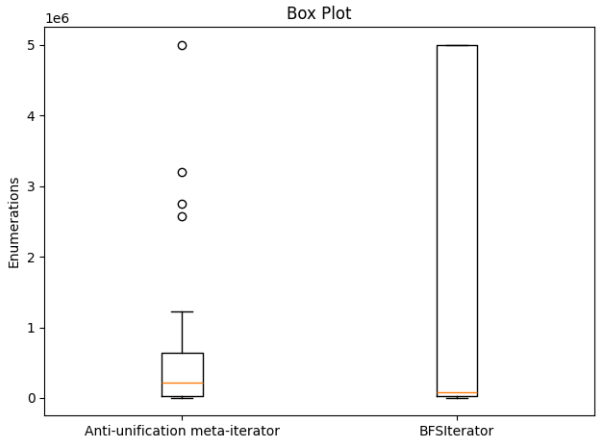

Figure 3: Comparison of Enumeration Distributions for BFSIterator and Anti-unification Meta-iterator on SyGuS SLIA Track

DFSIterator Comparison on SyGuS SLIA track

Out of the 100 problems, the Anti-unification Meta-iterator solved 23 problems, while the DFSIterator solved 21. There were 35 problems in which a common pattern was added to the DSL, and excluding the ones that both iterators did not solve, we arrived at a subset of 19 problems that should

be investigated. The subset comprises four problems solved only by the Anti-unification Meta-iterator, two problems solved only by the DFSIterator, and 13 problems solved by both.

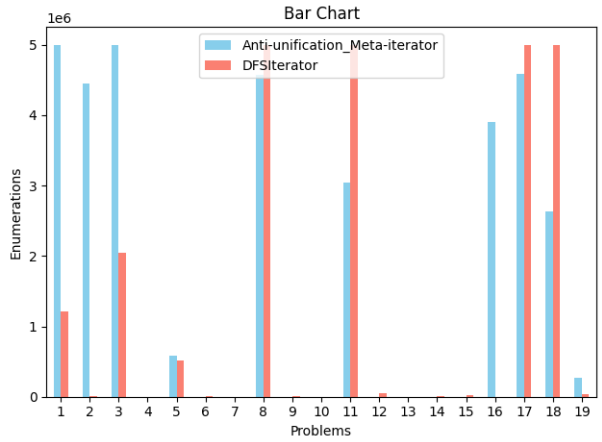

Figure 4: Enumeration Counts for DFSIterator vs. Anti-unification Meta-iterator on SyGuS SLIA Track

From this experiment, we can derive that iterators that explore the program space in a non-uniform style, such as the DFSIterator, which prioritizes exhaustive exploration, perform worse when used as sub-iterators for the Anti-unification Meta-iterator. Compared to iterators that employ a more uniform exploration, such as the BFSIterator, the downside of inserting a new rule into the DSL that is not useful in finding a correct solution is much higher. This increases the likelihood that the Anti-unification Meta-iterator will perform worse than its sub-iterator.

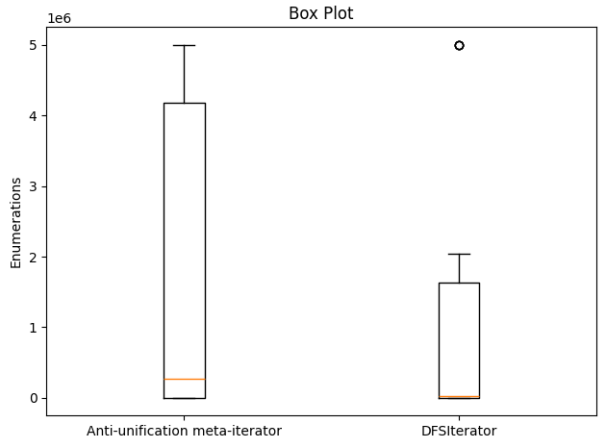

Figure 5: Comparison of Enumeration Distributions for DFSIterator and Anti-unification Meta-iterator on SyGuS SLIA Track

BFSIterator Comparison on SyGuS BV track

Out of the 250 problems, the anti-unification meta-iterator solved 26 problems, while the BFSIterator solved 25 problems. Not considering the problems for which the Anti-unification Meta-iterator found no common pattern, there will be 230 problems. By eliminating the problems that both iterators failed to solve, we have 18 problems. The subset comprises seven problems solved only by the Anti-unification Meta-iterator, six problems solved only by the BFSIterator, and five problems solved by both.

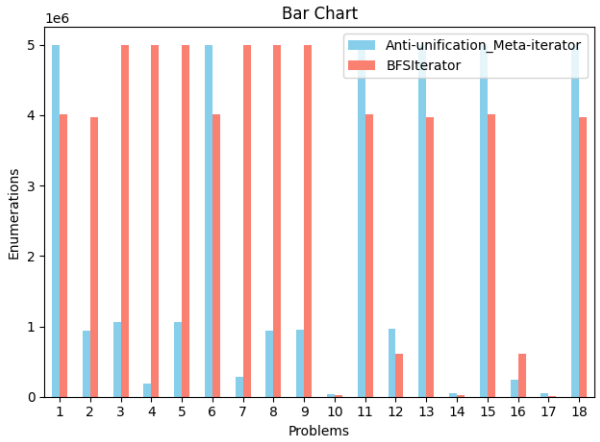

Figure 6: Enumeration Counts for BFSIterator vs. Anti-unification Meta-iterator on SyGuS BV Track

This experiment aims to assess the Anti-unification Meta-iterator's capabilities on more complex problems, characterized by a larger number of examples compared to the SyGuS SLIA track. From the evaluation results, we can conclude that the performance of the Anti-unification Meta-iterator is situational. The number of problems where the Anti-unification Meta-iterator required fewer enumerations is almost equal to the number where the BFSIterator performed better. However, this test suite also highlights the potential of the anti-unification approach in the program synthesis domain. Notably, when the Anti-unification Meta-iterator

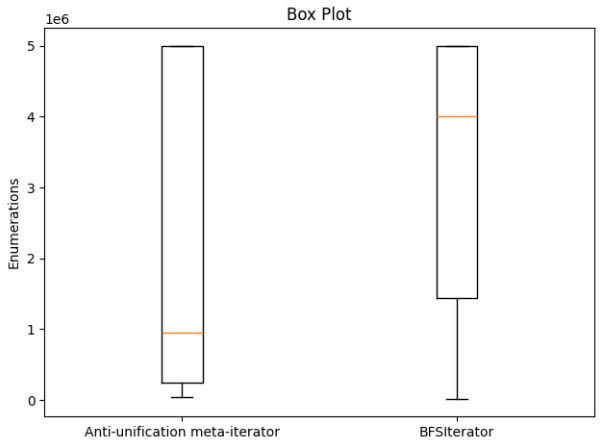

Figure 7: Comparison of Enumeration Distributions for BFSIterator and Anti-unification Meta-iterator on SyGuS BV Track

yields better results, the number of enumerations decreases by more than a factor of two, demonstrating its significant efficiency gains in specific scenarios.

Responsible Research

Ensuring the reproducibility of scientific research is fundamental for validating results and fostering further advancements. To facilitate this endeavor, this study offers a clear overview of all aspects of the setup, including software configuration, data sources, and algorithm parameters. This information has been meticulously documented in this paper's experimental setup and results section. Furthermore, the study offers a transparent view of the algorithm by providing a detailed description that motivates each decision.

The algorithm's source code will also be integrated into Herb.jl framework to facilitate the reproducibility of the experiments presented in the study. Herb.jl is an open-source project developed to create a unified and universal framework that lessens the burden of rewriting tools from scratch when researching software in the domain of program synthesis.

Conclusion and Future Work

This study explores the efficiency of creating a component library out of anti-unification to improve DSL-based program synthesizers. This paper seeks to show the potential of an anti-unification meta-iterator as an optimization by conducting a comparative analysis with other program iterators.

The approach presented in this study could still be vastly improved. A recurring problem that impairs the efficiency of the algorithm is the existence of redundant programs that satisfy user specifications by chance. This type of program are considered for the process of anti-unification, however the common patterns they provide do no benefit the process of synthesis, rather they they increase needlessly increase the DSL size leading to longer runtimes. An example for this behaviour is highlighted in the experimental setup and results section.

Another significant addition that could benefit the algorithm would be the detection of equivalent programs. Finding programs where one achieves the same user specification while doing more steps than required is frequent for complex solutions. For example, one '0' could be replaced in a problem with '(0 + 0)'. Applying anti-unification to such programs will lead to overly specific patterns that could not benefit the program synthesizer. A study that manages to solve this problem is Babble[3]. However, a domain-specific equational theory is required to approach this issue.

Lastly, collecting programs and applying anti-unification are processes that require a lot of computational power, especially for problems that are composed out of a combination of easy and hard to solve user-specifications. Discovering a way to improve this aspects through dynamic programming would improve the runtime of the algorithm.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[2] Peter E. Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Anti-unification algorithms and their applications in program analysis. In Andrei Voronkov Amir Pnueli, Irina Virbitskaite, editor, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–424. Springer, 2010.

[3] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. babble: Learning better abstractions with e-graphs and anti-unification. *arXiv preprint arXiv:2212.04596v1*, 2022.

[4] David M. Cerna and Temur Kutsia. Anti-unification and generalization: A survey. *arXiv*, June 2023.

[5] Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104), 2017.

[6] 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. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, pages 835–850, Virtual, Canada, 2021. ACM.

[7] Sumit Gulwani. Programming by examples (and its applications in data wrangling). *Microsoft Corporation*, 2016.

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

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

[10] G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. 1970.

[11] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, pages 135–151. 1970.