TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

Example-Based Synthesis using Context-Sensitive E-Graph Saturation

Matteo BERTOROTTA

Supervisors:
Dr. Sebastijan DUMANČIĆ
Dr. Andreea COSTEA
Tilman HINNERICHS, MSc.

19th September 2025

Delft University of Technology

Example-Based Synthesis using Context-Sensitive E-Graph Saturation

Master's Thesis in Computer Science

Algorithmics group / Programming Languages group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Matteo Bertorotta

19th September 2025

Author
Matteo Bertorotta

Title
Example-Based Synthesis using Context-Sensitive E-Graph Saturation

MSc presentation
September 26st, 2025

Graduation Committee

Dr. Sebastijan DUMANČIĆ	Delft University of Technology (chair)
Dr. Andreea COSTEA	Delft University of Technology
Dr. Jesper COCKX	Delft University of Technology

Abstract

Program synthesis is the task of constructing a program that satisfies specified constraints. One popular formulation of program synthesis is example-based synthesis. Here, the synthesizer attempts to find a program in a specified domain that satisfies a set of input-output examples. Enumeration is the most common approach to finding the desired program. However, the exponentially growing search space makes this infeasible. The size of the domain can be largely attributed to its inefficient representation. Often, programs are only syntactically distinguished, meaning programs that behave the same are seen as different. We introduce Context-Sensitive E-Graph Saturation, a novel method that limits the search space to programs that solve at least one of the provided examples. This allows focusing only on programs that behave similarly to the desired one. Crucial is finding contextual equivalences for each example over a generated termset. These equivalences allow generating many solutions for each individual example. A program in the intersection of these programs solves all examples. In experiments on a subset from SyGus SLIA, our method solves the problems that enumeration solves, but not more. These results highlight a trade-off: with a small termset, the discovered equivalences are often too limited to capture the relationships needed to find a universal solution. Conversely, increasing the termset size quickly leads to inefficiency. To address this, we propose a strategy for constructing a more expressive yet small termset, enabling our method to solve a broader range of synthesis problems.

Preface

The pages that follow represent a year of dedicated work: my Master's thesis in Computer Science. This research was carried out in the PONY lab of the Algorithmics group of Delft University of Technology under the supervision of Dr. Sebastijan Dumančić, Tilman Hinnerichs, MSc, and Dr. Andreea Costea from the Programming Languages group.

Program synthesis is notoriously complex, with an exponential growth in the number of programs and the intrinsic difficulty in bringing this down. A key observation is that there are vastly fewer behaviorally distinct programs than syntactically distinct ones. It was in this context that we discovered Ruler, a system capable of finding important equivalences within a domain of programs. The insight that such a system could be used to filter the search space from behaviorally uninteresting programs marked the birth of the idea for this thesis.

I would like to express my deepest gratitude to all my supervisors. Their guidance provided not only invaluable insights and the ability to apply years of knowledge, but also the encouragement and confidence to persevere when progress was slow.

I would also like to thank my family and closest friends—you know who you are—for their unwavering support and the time they dedicated to helping me with many aspects of this thesis. Thank you from the bottom of my heart.

Matteo Bertorotta

Delft, The Netherlands
19th September 2025

Contents

Chapter 1

Introduction

Having a way to describe candidate solutions alongside a set of requirements gives you the key ingredients for a powerful idea: automatically generating solutions. Many systems exist that can create a solution that abides by requirements, freeing the user from the details of how to build the solution and letting them focus solely on what the solution should achieve. This is the fundamental idea behind program synthesis.

Program synthesis is the task of constructing a program that satisfies specified constraints [1]. The power lies in the generality of this principle. In essence, anything can be described as a program. All that is needed is a language to express candidates and a mechanism to test them against constraints. From there, the search can be automated. Beyond its traditional association with computer programs and mathematical expressions, the usage of program synthesis ranges from domains such as performing auto-completion in Excel [2], to synthesizing molecules [3], and performing genome compression [4].

One popular formulation of program synthesis is example-based synthesis. Here, the synthesizer attempts to find a program in a specified domain that satisfies a set of input-output (IO) examples [1]. The program should return the corresponding output for each input from the list of examples. This paradigm is particularly powerful as it abstracts the need for users to specify how a program should work with formal specifications, focusing instead on what it should achieve. However, this ease of specification comes at a cost: IO examples are inherently less constraining; thus, example-based synthesis cannot guarantee that the synthesized program is exactly the one the user wants.

Enumeration is the most common approach to finding the desired program. With this approach, the program synthesizer systematically enumerates programs from the domain until it finds a program that meets the specified constraints [1]. Once such a program is found, we say that said program satisfies the constraints.

Example 1. Take the absolute function `|x|` defined as:

```
x >= 0 ? x : -x
```

which returns the magnitude of `x`. IO examples might map `-5` to `5` and `5` to `5`. The enumerator keeps enumerating programs, such as `x`, `-x`, `1 · x`, `-x+x`, until it finds a program that satisfies all the examples, like the desired absolute function. ⌟

Most approaches to program synthesis search through the often infinite set of all possible programs that a grammar can define. In theory, this makes it possible to find a program to solve any problem. In practice, however, the search space grows exponentially with program size, making exhaustive search infeasible even with techniques to guide it more efficiently [5, 6].

The size of the domain can be largely attributed to its inefficient representation. Often, programs are only syntactically distinguished, meaning programs that behave the same are seen as different. Therefore, those programs do not prevent each other from being considered. Ideally, once a program is identified as incorrect, programs that behave the same should no longer be considered. However, two programs can only be shown to behave equivalently once both have been fully constructed.

Example 2. Take the programs `0` and `-x+x` and assume we aim to find the absolute function. Once the synthesizer knows `0` is not the solution, ideally, it should not see `-x+x` as a potential candidate and never enumerate it. ⌟

This notion of behavioral equivalences [7] also affects the synthesis of vastly different types of programs like the ones discussed before [8, 4]. Therefore, if this problem can be solved, it will impact all these domains.

Furthermore, most programs in a domain do not satisfy any of the constraints. Most candidates are, therefore, uninteresting. As a result, much of the computational effort is wasted evaluating programs that, from the outset, have little chance of satisfying even a single example.

To solve both these problems, we would like to focus on programs that behave similarly to the one we seek. This allows us to focus on interesting programs and stops us from looking at different versions of uninteresting programs. The question then naturally becomes:

How can we limit the search space to programs that behave similarly to the one desired?

We propose a novel method: Context-Sensitive E-Graph Saturation. Think of each IO example as defining its own "universe" of programs that work for that example. We solely explore each such universe and skip (thereby pruning) uninteresting programs. Then, we intersect these universes to find a program that works for all examples. At a high level, the method proceeds in three stages.

The first stage focuses on analyzing each IO example individually. For each example, using a system named Ruler [9], we find a set of contextual equivalences over a termset that hold specifically under the conditions of that example. These equivalences define a set of transformations that identify behaviorally similar programs for the current example input. As these equivalences are context-sensitive, they produce behaviorally diverse yet example-consistent programs.

Example 3. Take the absolute function and the program `-x`. Normally, these programs are not equivalent. However, they behave identically on negative inputs, which can be expressed with contextual equivalences. ⌟

In the second stage, we use these equivalences to explore the space of programs that solve each individual example. This key step allows us to expand a small space of programs that satisfy said example to a large space of programs that solve it, thereby focusing on programs that solve at least one example.

Finally, we identify a program in the intersection of these expanded sets, utilizing a new algorithm presented. A program that lies in the intersection, by definition, solves all IO examples and thus satisfies the constraints.

Crucially, for all three stages, we use Equivalence Graphs (E-Graphs). E-Graphs allow us to efficiently represent and explore a vast space of program equivalences and to expand this space with new equivalences.

Recent research pointed out a significant bottleneck in Ruler, making it unable to efficiently find equivalences over many programs. Nonetheless, as large programs are made from smaller ones, equivalences theoretically only need to be found over a small set of programs. However, a larger set does allow finding more equivalences, some of which might be crucial. Thus, it remains valuable to investigate whether a large set can improve effectiveness in practice. To this end, several optimizations to Ruler are introduced. This brings us to the following research questions:

- **RQ1:** Does CSES perform competitively on real-world problems in relation to constrained enumeration?
- **RQ2:** Do the improvements on the bottleneck of Ruler improve efficiency?
- **RQ3:** Do the improvements on the bottleneck of Ruler help solve more problems with CSES?

We benchmark our method using a subset of problems from the SyGus SLIA domain, using tasks of varying difficulty. This allows us to assess how CSES performs across both simple and complex real-world synthesis problems.

We find that CSES only solves problems that enumeration also solves. While enumeration is consistently faster, CSES can complete execution in cases where enumeration times out. Furthermore, our improvements allow solving and terminating on more problems. However, CSES still fails to handle a larger termsets—as Ruler remains a significant bottleneck—and remains limited to problems enumeration solves too. The equivalences found over the terms in the small termset generated do not capture the key equivalences required to find a universal solution. Finally, an idea is suggested to acquire a small yet powerful termset.

Chapter 2

Background

This chapter provides the necessary background on program synthesis in general, focusing on example-based synthesis. We also explore rewrite rules, E-Graphs, and a tool that automates the creation of rulesets named Ruler, all of which are crucial in our approach.

2.1 Program Synthesis

Program synthesis [1] is the task of constructing a program that satisfies constraints describing what the program should do [1]; they are declarative. Interestingly, this differs from a compiler, which translates an already-written program to another language, and thereby uses a specification of how the program works rather than a specification of what it should do.

Program synthesis requires three elements: a Program Space, a set of constraints, and a search procedure. We will now explore these three elements.

2.1.1 Program Space

The synthesizer looks for a program in a provided domain: the Program Space $\mathcal{P}$. This domain consists of all programs that can be considered. Often, such a domain is infinite in size and consists of all programs that abide by some collection of rules that define their structure, or more formally, their syntax.

A common approach to describing the syntax of a language is with a Context-Free Grammar (CFG). For the purposes of explanation, we simplify a CFG to be a collection of terminal symbols Σ, i.e., the simplest programs, and a collection of production rules R that describe how a program can be made from smaller programs. Thus, given a CFG $G = (\Sigma, R)$, the programs in the Program Space $\mathcal{P}$ defined by G are all programs that can be constructed by G (see Definition 1). Basically, we create new programs by putting all our current programs in all possible orders into our production rules.

Definition 1 (Program Space)**.** Let $G = (\Sigma, R)$ be a CFG and let $\mathcal{P}_0 = \Sigma$. Then

$$\mathcal{P}_{n+1} = \mathcal{P}_n \cup \{\, r(p) \mid p \in \pi(\mathcal{P}_n),\ r \in R_p \,\}.$$

Here, $R_p \subseteq R$ denotes the set of production rules in R that can be applied to the collection of terms p, and $\pi(A)$ denotes all permutations of A. We define the entire Program Space $\mathcal{P}$ as $\mathcal{P}_\infty$. ⌟

Example 4. Take the CFG in Figure 2.1. The set of terminal symbols consists of the numbers and the variable `x`, $\mathcal{P}_1$ introduces terms like `-x` and `x>=0`, and $\mathcal{P}_2$ adds terms like `-x+x` and `x >= 0 ? x : -x`. Finally, $\mathcal{P}$ contains all terms that can be created. ⌟

```
Element = Num
Element = Bool
Element = Bool ? Element : Element
Num = 0 | 1 | 2 | 3 | ...
Num = x
Num = -Num | Num + Num | Num - Num | Num * Num
Bool = Num >= Num | Num < Num
```

Figure 2.1: An example CFG with numbers, a variable, Booleans, some standard numerical and relational operators, and an if statement.

2.1.2 Constraints

Now that we know where to look for the program, we need to specify what behavior we want the program to have. In other words, we need a way of describing what output corresponds to what input. These specifications are formed by constraints and can be complete or incomplete.

A complete specification describes exactly what a program should return for any input it gets. Writing such a specification can be as hard as writing the program itself. I will not focus on such constraints in this thesis.

An incomplete specification is not required to specify every part of the program, with the trade-off that there might be many undesired programs that also satisfy it. Example-based synthesis is a program synthesis problem (see Problem 1) that makes use of an incomplete specification. It consists of a collection of input-output (IO) examples, where, for each collection of inputs $\vec{x}$, the desired program should return the corresponding output y.

Problem 1 (Example-based synthesis)**.** Given IO examples $E = \{(\vec{x}_i, y_i)\}_n$ and a Program Space $\mathcal{P}$, find a program $p \in \mathcal{P}$, s.t. $\forall(\vec{x}, y) \in E.p(\vec{x}\ldots) = y$. ⌟

In general, a great heuristic for program synthesis is program size, as searching for the smallest correct program prevents overfitting. When the provided examples are sufficiently rich—meaning they capture the behaviorally distinct cases of the program—the smallest consistent program is often also the intended one.

Example 5. An example specification in the form of IO examples that describe the absolute function is:

```
[(-20,20), (-5,5), (0,0), (5,5), (20,20)]
```
(2.1)

However, besides the absolute function, an if-statement with a case for each input also satisfies these constraints. And if we were to extend the domain to allow fractions, so would the polynomial $0.21x^2 - 0.0004x^4$. Although every such program satisfies the examples, the absolute function is preferable because it is the smallest one. ⌟

2.1.3 Search Procedure

To find a program in the Program Space that abides by the specified constraints, we need some type of search. Most program synthesis algorithms use a form of enumeration as the search technique. Enumeration entails systematically exploring the Program Space in some specified way. Naturally, this approach has an exponential complexity, as the number of programs grows exponentially with their size.

With Breath-First enumeration, or bottom-up enumeration, programs are considered in order of size. We first explore all programs in $\mathcal{P}_0$, then the programs in $\mathcal{P}_1 - \mathcal{P}_0$ from smallest to largest, etc. Breath-First enumeration naturally achieves the property of finding the smallest successful program, as it explores the Program Space in order of increasing size.

Example 6. With the grammar from Figure 2.1, and the IO examples from Equation 2.1, BFS enumeration enumerates the constants and variables first: `0`, `1`, $\cdots$, `x`. It then enumerates programs combining these expressions, like `-x` and `x>=0`, and finally programs combining those expressions, like `x >= 0 ? x : -x`. As this function solves all examples, the enumeration comes to an end. ⌟

2.2 Rewrite Rules

A rewrite rule specifies mutual replicability between two terms (see Definition 2). Given a sound rewrite rule, the left and right sides interpret to the same final value for any identical substitution of their parameters, or free variables, and can thus be substituted by each other. Two terms that interpret to the same value are said to be behaviorally equivalent.

Definition 2 (Rewrite rule)**.** For any domain of terms $\mathcal{P}$ and terms $l, r \in D$ with free variables $\vec{x}$, let

$$\forall \vec{x}. l \leftrightarrow r$$

be a rewrite rule. ⌟

Definition 3 (Validity of a rewrite rule)**.** A rewrite rule $l \leftrightarrow r$ is said to be sound iff l and r are behaviorally equivalent for any substitution of $\vec{x}$. ⌟

From now on, we will assume universal quantification of free variables and just write $l \leftrightarrow r$.

Example 7. Take the terms `x` and `x+0`. No matter the value or expression we substitute for `x`, these expressions always return the same value. Therefore, $x \leftrightarrow x + 0$ is a sound rewrite rule. ⌟

As the variables of rewrite rules can represent any final value, these variables function as metavariables. In other words, the variables can be substituted by any term. If a rewrite rule r_1 implies another rule r_2 s.t. $r_1 \neq r_2$, r_1 is said to be stronger than r_2. A stronger rewrite rule is preferred, as it can prove more equalities.

Example 8. Using $x \leftrightarrow x + 0$, we can rewrite the term `2x` both to `2(x+0)`, matching on `x`, and `2x + 0`, matching on `2x`. Note that $x \leftrightarrow x + 0 + 0$ is also a valid rewrite rule, but is implied by $x \leftrightarrow x + 0$ and is therefore weaker. ⌟

2.2.1 Rewrite Ruleset

Individual (simple) rewrite rules over the same domain can be combined into a ruleset, forming a sophisticated set that can be easily extended and maintains equivalences between rewritten expressions. Smaller rulesets are desirable as fewer rules have to be matched against a term, making the process of using the rules faster. Furthermore, general, orthogonal terms are desired as these expand the number of terms that can be derived after a set number of rule applications.

2.3 Equivalence Graphs

While individual rewrite rules let us replace terms one step at a time, we often want to explore many possible rewrites at once. Doing this naively can cause an explosion in the number of terms to track. Equivalence Graphs (E-Graphs) are designed to solve this problem. Instead of storing every term separately, E-Graphs group terms together that are known to be behaviorally equivalent. They compactly represent and efficiently compute equivalences over a given set of terms implied by a collection of rewrite rules. These efficiencies are achieved with union-find data structures [10] and hashconsing [11], which allow E-Graphs to represent exponentially many terms.

2.3.1 Structure

An E-Graph (see Definition 6) represents terms using E-Nodes (see Definition 4) and represents equivalence relations by grouping behaviorally equivalent E-Nodes into E-Classes (see Definition 4). E-Graphs may contain cycles, which allows them to represent an infinite set of terms.

Definition 4 (E-Node)**.** An E-node is a combination of a constant symbol c, together with a (potentially empty) ordered list of arcs, each pointing to an E-Class, that form its parameters $\mathcal{A}$. If $\mathcal{A}$ is empty, c is either a constant or variable from the domain. Otherwise c is a function symbol and the E-Node represents all combinations of $c(t_1, t_2, \cdots, t_n)$ where t_i is a term represented by the E-Class pointed to by arc $a_i \in \mathcal{A}$. ⌟

Definition 5 (E-Class)**.** An E-Class is a collection of E-Nodes and represents all terms represented by its E-Nodes. Every term represented by the same E-Class is represented as equivalent. ⌟

Definition 6 (E-Graph)**.** An E-Graph is a collection of E-Classes and represents all terms represented by its E-Classes. ⌟

A rewrite rule can be applied on an E-Graph (see Definition 7), creating new terms and equivalences.

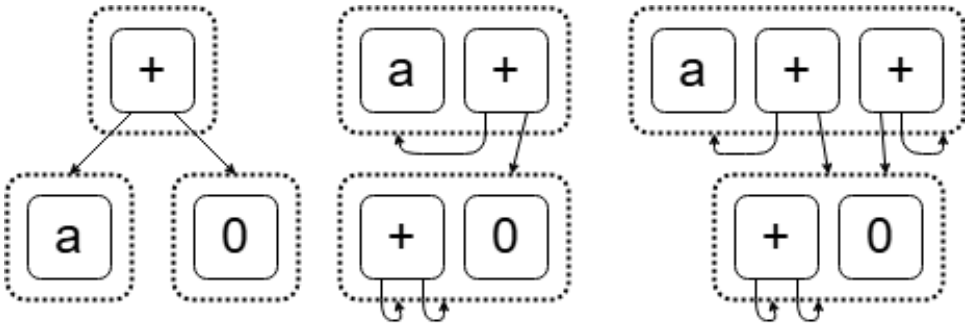

Figure 2.2: The first E-Graph represents the terms `a`, `0`, and `a+0` in separate E-Classes. When the rewrite rule $a \leftrightarrow a + 0$ gets applied, we get the second image where `a` and `a+0` are merged and the term `0+0` is added. Finally, when the rule $a + b \leftrightarrow b + a$ gets applied, we get the third picture where the new term `0+a` is added.

Definition 7 (Rewrite rule application)**.** For any E-Graph g and rewrite rule $l \leftrightarrow r$, let

$$g \xRightarrow{l \leftrightarrow r} g'$$

denote the application of rewrite rule $l \leftrightarrow r$ on g, resulting in a new E-Graph g'. ⌟

Example 9. Figure 2.2 shows three E-Graphs. We denote the first E-Graph as g, which represents the terms `a`, `0`, and `a+0` as three distinct terms. When we perform $g \xrightarrow{a \leftrightarrow a+0} g'$, we get the second E-Graph g'. This E-Graph represents `a` and `a+0` in the same E-Class to indicate their equivalence. It also adds the term `0+0` to the E-Class containing `0`. Both E-Classes have a cycle. Therefore, some terms also represented by the E-Graph are, for example, `(a+0)+0`, `(0+0)+0`, and `(a+0)+(0+0)`. In fact, E-Graph g' represents the equivalences over the infinite set of terms represented by the following grammar:

```
Num = a | Zero | a + Zero
Zero = 0 | Zero + Zero
```

Finally, we perform $g' \xrightarrow{a+b \leftrightarrow b+a} g''$, which adds the term `0+a` to the E-Class that represents `a`. Crucially, an E-Node does not represent its constant symbol. It represents a collection of equivalent complete programs. There are two E-Nodes with a $+$ as their constant in the same E-Class, as they represent inherently different programs. ⌟

2.3.2 Saturation

Equality Saturation[12] (see Definition 8) allows E-Graphs to function as rewrite engines. We start with an initial set of terms T, including a distinguished term $t \in T$, represented as an E-Graph with no information of equivalences. Therefore, every term in T starts in its own E-Class. Then, we repeatedly apply all rules from a given ruleset R to T to create a large set of equivalent terms. A user-provided cost function can then extract and return a desired term from the E-CLass containing t, used to replace t. Another use case of saturation is proving a behavioral equivalence. Instead of starting with one term $t \in T$, we can start with two distinguished terms $t_1, t_2 \in T$. After saturating T, we check whether their terms end up in the same E-Class, thereby proving their equivalence.

Definition 8 (Saturation)**.** For any E-Graph g and rewrite ruleset R, let

$$g \overset{R}{\Longrightarrow} g'$$

denote continuously applying the rules in R to g, until no rule in R changes g in any way, resulting in the saturated E-Graph g' based on R. ⌟

2.4 Iterative Rewrite Rule Inference

Creating rulesets manually is a complex process that requires domain experts. It is often done by trial and error and is therefore slow and error-prone. Generally, the following steps are performed iteratively:

1. Introduce terms from the domain into a termset T;
2. Select promising rule candidates from $T \times T$ to make C;
3. From the candidates in C, choose a set of valid rules to extend ruleset R.

Ruler [9] is a tool created to automate the creation of rulesets. It infers rewrite rules using iterative equality saturation and uses an E-Graph to represent the termset T. Ruler functions iteratively; A ruleset R is used to saturate T while R is being synthesized. We will now focus on the three steps performed in each iteration.

2.4.1 Introducing Terms with Characteristic Vectors

When a term gets added to T, it is assigned a characteristic vector (`cvec`) (see Definition 9). A `cvec` serves as a collection of output behavior examples.

Definition 9 (Characteristic vector)**.** A $\texttt{cvec}_n$ is a list of n outputs produced by the term. Constants always return the same value, namely themselves; thus, a $\texttt{cvec}_n$ of a constant contains n copies of said constant. A variable can return any value from the domain, thus, the $\texttt{cvec}_n$ of a variable contains n values chosen from the domain. The $\texttt{cvec}_n$ of a function is made by applying the operator on the `cvec` of its parameters. ⌟

Example 10. Take the first E-Graph from Figure 2.2 based on the grammar from Figure 2.1. The $\texttt{cvec}_3$ of the E-Class with `0` is $[0, 0, 0]$. The $\texttt{cvec}_3$ of `a` could be $[1, 2, 3]$. Then, the $\texttt{cvec}_3$ of `a+0` would be $[1 + 0, 2 + 0, 3 + 0] = [1, 2, 3]$. ⌟

The set T of terms is initially empty, and in each iteration i, new terms are added to T. In the i'th iteration, the added terms have i operators. This makes sure that any equalities over subterms have already been discovered for every term.

In the first iteration, with $i = 0$, all terms with 0 operators, i.e., all constants and variables, are added. This iteration thereby specifies all such unique singular terms. The second iteration, with $i = 1$, adds terms with one operation, like `x+0` and $\sum_i^n i$.

2.4.2 Selecting Candidates

We first perform $T \overset{R}{\Longrightarrow} T^*$, using the current (potentially empty) ruleset R. Then, we update T by merging the E-Classes that got merged in T^*, which makes sure we do not introduce new terms to the termset. Together, this process is named `run_rewrites`. Because terms in the same E-Class of T^* are proven equal, we do not need to consider pairs of terms from the same E-Class as candidate rules, as they are already proven to be equivalent with the current ruleset R. Consequently, we only need to consider one term, the canonical term, for each E-class in T when adding rules to the candidate set C. Doing this, we can ignore many potential candidate rules.

However, comparing the canonical term from each E-Class with every other such term would still take a lot of time. Therefore, we only create candidate rules between canonical terms with matching `cvecs`, as any rule between two terms with non-matching `cvecs` is already guaranteed to be invalid. Thus, all canonical term pairs with matching `cvecs` form the candidate set C.

Example 11. Following Example 10, we would add the rule $a \leftrightarrow a + 0$ to C, because both have $[1, 2, 3]$ as their $\texttt{cvec}_3$. ⌟

Algorithm 1: SHRINK

Input: Candidate rules C, rewrite rules R
Output: Pruned set of candidate rules $C' \subseteq C$
for $l \leftrightarrow r \in C$ **do**
  Add l and r to g `/* E-Graph g is initially empty */`
return $\{l \leftrightarrow r \in C \mid l$ and r are not equivalent in $\texttt{run\_rewrites}(g, R)\}$

2.4.3 Extending the Ruleset

Now we need to choose new rules from the candidate set. Ideally, this selection of rules is the smallest valid extension that can establish all valid equivalences implied by $R \cup C$. To achieve this, Ruler starts by selecting `step` rules from C according to a heuristic, checks their validity using a domain-specific approach, and adds them to a set K. Then, it uses them to `shrink` (see Algorithm 1) the remaining candidates in C with $R \cup K$. This process goes on until C is empty. Once C is empty, the process gets repeated with a smaller `step` size until `step` equals 1. When `step` is large, it can quickly trim C down, and when it is small, it can trim more exactly. By iteratively decreasing `step` to 1 and pruning C, both qualities are obtained. When `step` is 1, the selected rules get added to R.

`shrink`ing (see Algorithm 1) makes sure the remaining candidates cannot be derived from the ones that have already been considered. It adds all terms from the rules in C to an empty E-Graph g and performs `run_rewrites` with a supplied ruleset R, which saturates the E-Graph. Then, it goes over every rule in C again to see if both terms end up in the same E-Class, which implies the rule is proven by R.

2.4.4 A bottleneck in Ruler

When we dive into Ruler as an algorithm, we discover an interesting problem: the size of the candidate set forms a bottleneck to Ruler. Note that Ruler makes a candidate rule between all E-Nodes of each separate E-Class with the same `cvec`. The number of these rules grows quadratically with the number of terms. Furthermore, significantly decreasing the size of the candidate ruleset in some way, like

`shrink`ing, is required as saturation becomes infeasible with an increasing number of rewrite rules. However, `shrink` is a costly procedure that is at least linear in the number of candidate rules, and upper-bounded by the saturation [13]. Ruler shrinks the same candidate ruleset numerous times during the process of selection, which makes this process at least quadratic in the number of valid candidate rules and still upper-bounded by the saturation. Two works in literature address this very problem with Ruler. Isaria [14] is a system that automatically finds rewrite rules to generate vectorizing compiler transformations. When it uses Ruler, it encounters the problem that the candidate sets produced are large and grow fast, which slows the system down. It proposes a domain-specific solution to solve this problem that uses a heuristic cost to further prune candidates. Enumo [15] is a general rewrite rule generation system that tries to improve upon Ruler. It finds that Ruler can only handle "a few" iterations of adding terms before exponential growth makes it infeasible. To address this, it gives the possibility to focus the search in a domain-specific way.

Chapter 3

Related Work

The method explored in this thesis leverages a form of semantic information to reduce the search space originally purely based upon syntactical structure, by grouping programs together that behave the same on individual inputs. This principle can be expressed as a form of abstract interpretation, where each grouping is its own abstract domain. While abstract interpretation has not been used to reduce the search space, that is not to say that it has not been utilized at all in example-based synthesis. In this chapter, we will explore some of those strategies that use abstract interpretation to optimize program synthesis and how they compare to Context-Sensitive E-Graph Saturation. We will find two noteworthy differences: All these approaches utilize their abstraction to further specify an initial hypothesis, and all require the user to (partly) define the abstract domain used.

3.1 Morpheus and Neo: Guiding Search by Behavior

We start with two easy examples, Morpheus and Neo. Morpheus [16] is a synthesis algorithm designed to automate the synthesis of data transformation programs, specialized in tasks like table consolidation and transformation. It decomposes complex transformation tasks into smaller, manageable components, each associated with specific operators or functions. For instance, in a filter operation, the output table's rows cannot increase with respect to the input's rows, while the number of columns remains the same. These components are defined by user-provided specifications that enforce constraints on the input and output of the operators.

Because these specifications abstractly represent the behavior of operators, they, thereby, specify an abstract semantics. This abstraction is used when constructing a hypothesis. Abstractions from multiple operators get combined into a single formula with placeholders for intermediate tables, thereby creating an abstract representation of the hypothesis's behavior. Over time, such a hypothesis gets filled in using an SMT solver that ensures the satisfiability against given IO examples, rejecting any unsatisfiable hypotheses in the synthesis process.

Neo [17] enhances this synthesis process by introducing a conflict-driven learn-

ing technique capable of learning from past mistakes. When a generated program violates the specification, Neo identifies the root cause of the conflict to refine the specification, making it distinguish the detected conflict, which prevents similar mistakes in the future.

3.2 Blaze and ATLAS: A More Refined Approach

Blaze [18] is a program synthesis tool that uses an approach they call counter-example guided abstraction refinement. Like Neo, this method iteratively refines an initial abstraction of the desired program's behavior with found counterexamples. However, Blaze represents the abstract program using an Abstract Finite Tree Automata (AFTA). A state in an AFTA corresponds to abstract values from the program domain, and a transition naturally corresponds to a modification of such a value. Any program accepted by an AFTA abides by its specification. Thus, a program that abides by the AFTA yet fails the provided examples serves as a counter-example to optimize the AFTA. Interestingly, both AFTAs and E-Graphs serve to represent the behavior of programs. However, where AFTAs excel at capturing the behavior of one abstract program, E-Graphs capture the (abtract) behavioral similarities between multiple programs [19].

Crucially, the abstract semantics of the language is provided by a domain expert. Meaning, the domain expert provides a collection of predicates that may be used in the abstractions from the synthesis procedure, together with the abstract semantics of each language construct. Importantly, each function needs to have an abstract return value given abstract parameters. Clearly, this is a time-sensitive task.

ATLAS [20] is made to automate this process. It can learn and improve its abstract semantics with multiple synthesis programs over the same domain. While making Blaze more versatile, this process involves tree interpolation and solving second-order constraints, which is computationally expensive. It therefore does not adapt to new problems when the semantics are obtained from training. As Pallabi Sree Sarker likes to say [21], once they are learned, they are learned. This makes it less adaptive to a new synthesis environment. Furthermore, to find a useful abstraction for the domain, the training synthesis problems need to be representative enough, which requires a domain expert when creating them.

3.3 Simba: Abstracting the Other Direction

Simba [22] is a forward-backward abstract interpretation approach to program synthesis. Besides abstracting a program's output when given a certain input, which all discussed approaches do, it also constructs abstract inputs given outputs. This abstraction from both perspectives allows it to refine the search space from both directions simultaneously.

Simba implements this methodology through a combination of forward and backward abstract semantics. It first generates partially-implemented programs

using a top-down search procedure. Then, for each partial program with missing expressions, a forward analysis computes approximated invariants of the program's output behavior given the input examples. Then, a backward analysis determines the necessary conditions for the missing expressions in the program given the output examples. These analyses have synergy as both can be used to refine the other iteratively.

However, like Blaze, Simba's approach has the crucial limitation that it relies on highly precise abstract domains. Using Simba for a new domain is therefore non-trivial, which makes it not easily adaptable to other domains.

3.4 Absynthe: A Specializable Approach

Another approach that uses abstract interpretation to guide synthesis is Absynthe [5], which is unique because of its ability to combine abstract domains. This flexibility creates a trade-off that allows to have more expressive domains that prune more programs yet require more time.

Absynthe's abstract interpretation approach is quite general. It involves defining an abstract domain and an abstract interpreter for the desired program, which guide the synthesis process. It starts from the abstract return value of the function and, at each step, further concretizes the abstraction with one that satisfies the original abstraction. This continues until a concrete program is formed, which gets tested on the examples. It terminates once it finds a valid candidate.

3.5 Reflection

All approaches we have seen, except ATLAS, rely on domain-specific semantic specifications that need to be defined. In contrast, the approach explored in this thesis is designed to be domain-agnostic, enhancing its applicability across various program synthesis tasks. All that is required is a specification of the language and the examples. The abstraction follows naturally from those. Furthermore, all methods explored start from an initial abstract state that represents the current program, which gradually gets specialized towards the desired program using semantic specifications. Thus, these approaches can only guide the search using interesting behavior. The approach in this thesis instead learns behavioral similarities per example to create an abstract program space where all programs behave identically on said example and solve it. The program then follows directly from the intersection of these spaces. Finding these spaces can thus be seen as a form of pruning instead of guiding, with the clear advantage that once such a space is discovered, it does not need to be rediscovered and can be reused for other problems. It can, however, not be improved with new problems. In this way, our approach is most similar to ATLAS.

Chapter 4

Problem Statement

The problem we focus on is simply the general example-based synthesis problem defined in Chapter 2 (see Problem 1). Our attention is directed toward the characterization of the Program Space $\mathcal{P}$ (see Definition 1). Normally, $\mathcal{P}$ consists of all programs that can be generated from some grammar G. However, we want to focus solely on programs "close to" the desired program p that solves all examples. Such a criterion is naturally hard to define. Here, we deem a program interesting when it solves at least one example. The set of all interesting programs can accordingly be defined as in Definition 10.

Definition 10 (Refined program space)**.** Given IO examples $E = \{(\vec{x}_i, y_i)\}_n$ and Program Space $\mathcal{P}$, let

$$\mathcal{P}^* = \{p \mid p \in \mathcal{P}, \exists(\vec{x}, y) \in E.p(\vec{x}\ldots) = y\}$$

be denoted as the refined program space given E. ⌟

This refined Program Space $\mathcal{P}^*$ focuses on the subset of $\mathcal{P}$ where each program is a partial solution. Note that, by definition, these programs are observationally equivalent on at least one example to a program p that solves all examples. We now define a notation for such observational equivalence on a specific example and will refer to it as contextual equivalence (see Definition 11). Two programs are contextually equivalent on inputs $\vec{x}$ iff they produce the same output when run on the inputs in $\vec{x}$.

Definition 11 (Contextual equivalence)**.** Let $\vec{x}$ be a collection of inputs, and let p and q be programs. Then,

$$p \equiv_{\vec{x}} q$$

denotes the contextual equivalence between p and q on inputs $\vec{x}$. ⌟

For ease of notation, from now on we will write $\equiv_x$ when $|\vec{x}| = 1$ and $x \in \vec{x}$.

Definition 12 (Contextual equivalence validity)**.** Let $p \equiv_{\vec{x}} q$ denote a contextual equivalence between programs p and q on inputs $\vec{x}$. This contextual equivalence is said to be valid iff

$$\forall \vec{v}.p(\vec{x}\ldots) = q(\vec{x}\ldots)$$

where $\vec{v}$ ranges over all free variables in p and q except the ones in $\vec{x}$, and equality denotes behavioral equivalence. ⌟

Example 12. Take the terms `x-4` and `2x+1`. Generally, these two terms are clearly not equivalent. However, when we specify the input to `-5`, both return `-9`. Thus, they are contextually equivalent under `-5`, resulting in $x - 4 \equiv_{-5} 2x + 1$. ⌟

Contextual equivalence $\equiv_{\vec{x}}$ creates a partition $\mathcal{P}_{\vec{x}}$ of the Program Space $\mathcal{P}$ where programs in a class $\mathcal{P}^t_{\vec{x}}$ of the partition $\mathcal{P}_{\vec{x}}$ are contextually equivalent on inputs $\vec{x}$ and where the class $\mathcal{P}^t_{\vec{x}}$ contains term t. This is due to contextual equivalence being an equivalence relation by definition. Now we can redefine the refined Program Space $\mathcal{P}^*$ using the collections of contextually equivalent programs per example (see Definition 13).

Definition 13 (Refined program space 2)**.** Given IO examples $E = \{(\vec{x}_i, y_i)\}_n$ and Program Space $\mathcal{P}$, let

$$\mathcal{P}^* = \bigcup_{i=1}^{n} \mathcal{P}^{y_i}_{\vec{x_i}}$$

⌟

This definition exposes an interesting aspect of the refined Program Space: all programs in $\mathcal{P}^*$ are behaviorally equivalent to the desired program p when considered modulo the IO example they solve. In consequence, we can represent the set of correct programs as:

$$S = \bigcap_{i=1}^{n} \mathcal{P}^{y_i}_{\vec{x_i}} \tag{4.1}$$

This characterization highlights that the synthesis problem can be viewed as identifying the intersection of all contextual equivalence classes that individually satisfy each example. In this sense, synthesis reduces to locating the common behavioral core across all examples. Based on this perspective, we reformulate the synthesis task as a problem of discovering this contextual equivalence class for each example (see Problem 2).

Problem 2 (Finding the refinement)**.** Given IO examples $E = \{(\vec{x}_i, y_i)\}_n$ and a Program Space $\mathcal{P}$, find, for every example $(\vec{x}, y) \in E$, the class $\mathcal{P}^y_{\vec{x}}$. ⌟

The method we will explore tackles this newly defined problem. It still solves the original example-based synthesis problem, yet not by focusing on conventional enumeration over the Program Space $\mathcal{P}$. Instead, it tries to explore all interesting classes.

Chapter 5

Context-Sensitive E-Graph Saturation

This section describes the approach explored in this thesis to perform example-based synthesis efficiently, namely, Context-Sensitive E-Graph saturation. The goal is to explore the space of programs that solve at least one example, thereby skipping over numerous (semantically identical) uninteresting programs, pruning them from the Program Space. Algorithm 2 describes the Context-Sensitive E-Graph Saturation (CSES) algorithm, which is parameterized by:

- the IO examples that define a correct program;
- the grammar that defines the Program Space;
- a potential collection of constraints over the grammar;
- the maximum length of small terms enumerated;
- a potential collection of general rules over the domain;
- a potential collection of wildcard rules to extend a ruleset.

In short, we start by generating an initial termset consisting of programs that solve individual examples together with small terms generated from the given grammar. This is the termset over which we find our rewrite rules. Using Ruler (see Section 2.4), we obtain a collection of rewrite rules per example over the termset, where each such rule holds in the context of that example. For each such ruleset, we can enlarge the subset of our termset that solves the example corresponding to the ruleset by saturating it. Together, these steps give an approximate solution to Problem 2 and prune the initial Program Space to programs that solve at least one example. Finally, once we have these collections of correct programs per example, we try to find a program that lies in their intersection, and therefore satisfies all examples, thus solving Problem 1. First, we discuss an example usage of CSES.

Algorithm 2: CSES

Input: Examples E, grammar G, constraints $\mathcal{L}$, maximum length n, general rules R, wildcard rules $\mathcal{W}$
Output: A program $p \in G$ that solves all examples in E, or $\perp$ if no such program is found
$D_1 \leftarrow$ Enumerate programs from G up to length n
$D_2 \leftarrow \{$enumerate programs from G constrained by $\mathcal{L}$ to solve $e \mid e \in E\}$
$D \leftarrow D_1 \cup D_2 \cup \{\vec{x} \cup \{y\} \mid (\vec{x}, y) \in E\}$
for $(\vec{x}, y) \in E$ **do**
    $R_i, T_i \leftarrow$ Set variable $\texttt{cvec}_1$s to value in $\vec{x}$, run `ruler` on D with R
    $P_i \leftarrow$ extract sub-E-Graph from T_i with programs from E-Class with y
    $P'_i \xLeftarrow{R_i \cup \mathcal{W}} P_i$
return INTERSECTION(P'_1,$\cdots$,$P'_{|E|}$)

Then, we explore the part of the system designed to give an approximate solution to Problem 2. Next, we dive into the details of the Intersection. Finally, we discuss the implementation details of the system.

5.1 Motivating Example

Figure 5.1 shows an example usage of CSES. We are given a grammar over numbers, together with a set of examples:

$$[(0, 1), (1, 3), (2, 5)].$$

First, we enumerate programs from the grammar up to a depth of $n = 1$, giving us the constants, variable x, and all programs that use at most one operator. Then, we enumerate solutions to individual examples. Together, these programs form the termset. For each example, we find collections of programs in the generated termset that interpret to the same value on that example. This is represented by the value in the gray circles. We use this, together with a potential set of supplied general rules, to create a contextual ruleset for each example. Note that, for the first example, we find a contextual equivalence between x and $2x$. We saturate the solutions for each example with the ruleset we found for that example and a potential set of supplied wildcard rules. For the first example, this allows us to create $2x + 1$ from $x + 1$ using our discovered equivalence. Note that the other examples also find $2x + 1$ using their own contextual rules. Finally, we try to find a program in the intersection of the solutions for each example. We find $2x + 1$, which indeed solves all examples.

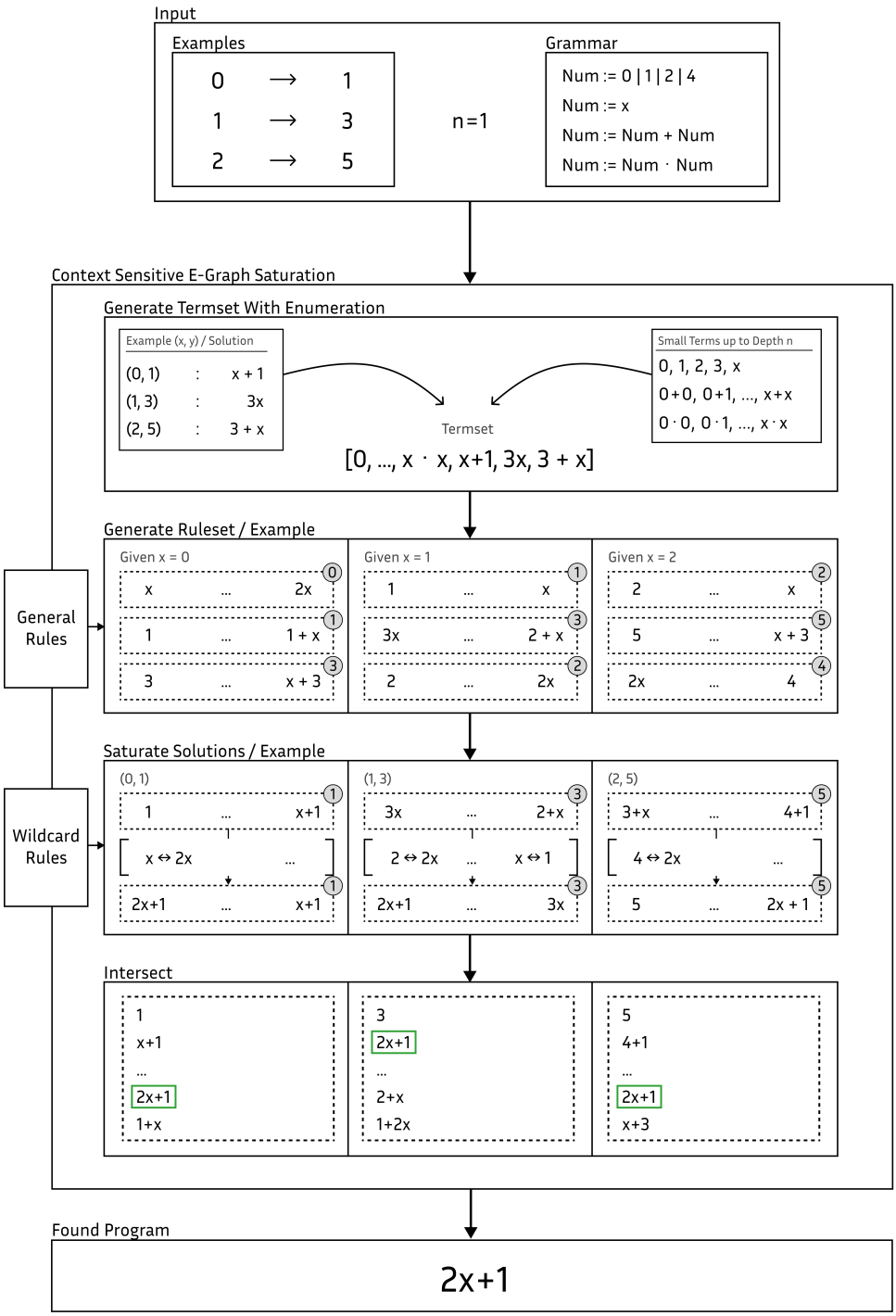

Figure 5.1: Example usage of CSES. The input consists of IO examples, a grammar, and number n. CSES generates a termset, consisting of small terms up to size n, and solutions to individual examples, enumerated from the grammar. Then, Ruler finds a ruleset over the termset for each example. This gets used to saturate the set of programs that produce the correct output under the example input, yielding a collection of solutions per example. Finally, these program collections get intersected, giving us the solution.

5.2 Solving the Problem

Our goal is to solve Problem 2. If we were given the complete partition $\mathcal{P}_{\vec{x}}$ for each example $(\vec{x}, y)$, our mentioned goal would be trivial and follow directly from the partitions. However, defining contextual equivalence $\equiv_{\vec{x}}$ for each example $(\vec{x}, y)$ for the entire Program Space $\mathcal{P}$ is clearly infeasible, as the Program Space is simply too large. Instead, we need some other approach to discover the classes $\mathcal{P}^{y}_{\vec{x}}$. We will now discuss how we can achieve this.

Assume that we already have a set of seed programs $P = \{p_1, p_2, \ldots, p_n\}$, where each program $p_i \in P$ solves the ith input-output example $(\vec{x_i}, y_i)$ from the given set of examples; that is, $p_i(\vec{x_i} \ldots) = y_i$ for all $1 \leq i \leq n$. Let $P_p = \{q \mid q \in \mathcal{P}, q \equiv_{\vec{x}} p\}$ be the set of contextually equivalent programs in $\mathcal{P}$, defined over all $p \in P$. Note that all programs in P_{p_i} solve example $(\vec{x}_i, y_i)$. Therefore, by definition, $P_{p_i} = \mathcal{P}^{y_i}_{\vec{x_i}}$. This shows that we can obtain $\mathcal{P}^{y_i}_{\vec{x_i}}$ by enlarging $\{p_i\}$ to P_{p_i}.

Starting from just a program p_i as an initial subset of $\mathcal{P}^{y_i}_{\vec{x_i}}$, we can use rewrite rules to enlarge it and approach P_{p_i}. Rewrite rules are especially suited for this as they allow us to use the additional context that the programs need only be equivalent on the inputs $\vec{x_i}$. Normal behavioral equivalence is not required. The difficulty, therefore, shifts from creating classifications to finding powerful rewrite rules. The subsequent subsections describe our approach to this task in detail.

5.2.1 Creating the Termset

We want to obtain a collection of rewrite rules able to enlarge a collection of programs that solve an example. For such a ruleset to be useful, it requires two things: equivalences over the behaviorally interesting parts of the solution program, and the ability to enlarge a specific subspace of programs. Using both kinds of rules, one can enlarge the behaviorally interesting parts until the solution program is found. This makes it possible to explore the vast space of programs that behave identically on a specified input targeted toward the solution program. To create these rewrite rules, we need an initial termset D over which the rules are formed. We will now discuss, for both requirements, a collection of terms over which such rules can be formed.

To have rules that enlarge a subspace, we need small terms, as large terms often contain many small terms [23]. Capturing equivalences over subexpressions allows us to rewrite large terms and explore their syntactically similar and behaviorally equivalent region. Generating small terms is trivial and can be done by enumerating all programs from the grammar up to a size n (line 3). Here, size is indicated by the number of operators. A larger n increases the rewrite power of the found rulesets but significantly decelerates the process of finding said ruleset as it exponentially increases the size of terms over which the rules need to be defined.

Having terms that capture behaviorally interesting parts of the solution program comes down to having a collection of non-trivial solutions to individual examples. A program for an individual example creates a starting point for that particular

behaviorally distinct part of the solution program. A trivial way to find partial solutions is to enumerate programs until each example has one that solves it. However, programs found in this manner often do not provide any more information than the example output itself. By using constraints in the enumeration specific to the domain, this can be overcome as much as possible (line 4). Finally, D is created by combining the small terms with all partial solutions and all example input and output values (line 5).

Example 13. Take the following program that solves the IO example (x_i, y_i), where e can be any expression:

$$\texttt{x == x ? } y_i \texttt{ : } e$$

This program is behaviorally equivalent to the program y_i on input, thus expresses no specific behaviorally distinct part of the solution program other than returning the correct output. ⌟

5.2.2 Obtaining the initial rulesets modulo example

Our goal is to obtain a collection of rewrite rules R_i over D for every example $(\vec{x_i}, y_i)$ that holds in the context of the example. For this purpose, we define contextual rewrite rules (see Definition 14).

Definition 14 (Contextual Rewrite Rule)**.** For any value n, let

$$l \leftrightarrow r \mid x \leftarrow n$$

be a rewrite rule (see Definition 2) where we can substitute every occurrence of x in l and r with n. ⌟

Definition 15 (Contextual rewrite rule validity)**.** A contextual rewrite rule $l \leftrightarrow r \mid x \leftarrow n$ is said to be valid iff $l \equiv_n r$. ⌟

Example 14. Take Example 12 where $x - 4 \equiv_{-5} 2x + 1$. This gives rise to the rewrite rule $x - 4 \leftrightarrow 2x + 1 \mid x \leftarrow -5$ with which we can rewrite $3x - 4$ to $4x + 1$:

$$\begin{aligned} 3x - 4 &\to 2x + x + x - 4 \\ &\to 2x + 2x + 1 \qquad\qquad (x - 4 \leftrightarrow 2x + 1 \mid x \leftarrow -5) \\ &\to 4x + 1 \end{aligned}$$

These programs behave the same on the input -5. Note that we also make use of some general rules like $nx \leftrightarrow (n-1)x + x$, which hold in any context. ⌟

A ruleset of contextual rewrite rules is special in the sense that it can find programs that are behaviorally equivalent in the context they are based on. Where we gain additional power in the form of larger applicability, we lose generality. These variables in a contextual rewrite rule are not metavariables as they are in normal rewrite rules. These variables thus cannot represent every expression, just their context. The terms `x-4` and `2x+1` are only equal in the context where `x` represents `-5`.

To obtain the desired rulesets R_i, we use Ruler with $\texttt{cvec}_1$s (see Definition 9). For every example, we specify the $\texttt{cvec}_1$ of the variable to its corresponding input from that example (line 7). By doing this, all equivalences found by Ruler are solely based on their example. We then obtain the desired rulesets R_i by running this specialized version of Ruler over the termset D for every example $(\vec{x_i}, y_i)$ (see line 7).

5.2.3 Saturating the Correct Classes

Because we specified Ruler based on individual examples, aside from obtaining the rulesets, we obtain a collection of programs P_i per example that satisfy it. Namely, the programs represented by the E-Class for which the $\texttt{cvec}_1$ equals the output value of the example. Because Ruler does not introduce new terms to the initial termset D, this will be the subset of D that solves said example.

Example 15. Take IO example $(-5, -9)$. After specializing Ruler on the input -5 such that x has $[-5]$ as $\texttt{cvec}_1$, and running it on a termset, the E-Class with a $\texttt{cvec}_1$ of $[-9]$ will contain programs that solve this example, such as `x-4` and `2x+1`. ⌟

As discussed, the rulesets found uphold contextual equivalence regarding the example they are based on. Therefore, when we use a ruleset on the programs that solve its example, the programs found will still solve the example. This allows us to focus solely on programs that solve at least one example.

Depending on the domain, we may desire to extend a ruleset R_i with a ruleset $\mathcal{W}$ containing wildcard rules (see Definition 17), thereby creating the ruleset $R_i^* = R \cup \mathcal{W}$ (see line 9). Generally, the reason to extend a ruleset is to enlarge the set of equivalent programs that can be found. As useful rule extensions are heavily domain-specific, they are part of the algorithm as a potentially provided parameter. Wildcard rules will be explained in detail when we dive into the intersection algorithm.

For each example $(\vec{x_i}, y_i)$, we extract the sub-E-Graph that contains all programs represented by the $\texttt{cvec}_1$ that equals $[y_i]$ to obtain P_i (line 8). Then, we saturate each P_i using its ruleset R_i^*, $P_i \overset{R_i^*}{\Longrightarrow} P_i'$ (line 9). The programs that form P_i' are the ones that end up in the same E-Class as the initial programs that solved the example and function as our approximation of $\mathcal{P}_{\vec{x_i}}^{y_i}$.

5.2.4 Optimizations to Ruler

As discussed in Subsection 2.4.4, Ruler's rule selection approach is expensive and grows quadratically with the number of candidate rules. For this domain, it becomes even worse. Normally, by verifying the initial set of candidate rules, a significant portion of rules can be removed and will never be considered again. However, in this domain, any rule found is valid by definition. Thus, all found candidates end up in the final candidate set. Furthermore, as we use $\texttt{cvec}_1$s, there are fundamentally fewer ways for `cvecs` to differ. For two $\texttt{cvec}_1$s to be equal, it suffices for them to have one—their only—value in common. As there are fewer different `cvecs`, groups of E-Classes with equal `cvecs` will be larger. Therefore, the number of candidate rules found increases. Lastly, the number of terms in the termset D we pass to Ruler grows exponentially with the maximum size of small terms n. Thus, in our domain, the complexity of selection, and thus of Ruler, grows exponentially with n. In comparison with enumeration, the exponential complexity shifts, therefore, to Ruler. We propose a selection of improvements that help address this exponential nature for our domain.

First of all, we can supply Ruler with a domain-specific general rewrite ruleset R, to which all R_i get initialized (see line 7). This ensures that Ruler starts by grouping every trivially equal term, like x and $x + 0$, together at each iteration. A single general rewrite rule has a contextual counterpart for each possible instantiation of its input. Therefore, this can significantly decrease the number of generated candidate rules. Furthermore, these general rules are more powerful than their contextual counterparts, as their variables can function as metavariables. However, as these general rules can target more terms, this also means that saturating with them might take longer than with contextual rules, and create a significantly larger final E-Graph.

The second improvement concerns a change to Ruler's rule selection approach. Ruler performs ten rule selection iterations starting with a rule selection amount of $\texttt{step} = 101$, decreasing the number of rules it selects per `shrink`ing round by 10 every iteration (see Section 2.4). Notably, this number does not change with regard to the size of the candidate set. We propose a rule selection approach where, instead, we iteratively select using $\texttt{step} = \lfloor \frac{|C|}{2} \rfloor$ until either $|C|$ does not change, or $|C| \leq 200$. Doing this, `shrink`ing gets done more aggressively with a compromise of a more difficult saturation each time. Once this approach comes to an end, we perform Ruler's standard selection procedure with a small alteration described next.

We modify Ruler's selection procedure by decreasing `step` only down to a minimum of 11 instead of 1. The motivation behind this change is that a step size of 11 is already sufficient to trim down a significant portion of rules, while reducing it further to 1 would force the procedure to linearly scan through the entire remaining candidate set with little chance of making a substantial impact.

For the next improvement, we observe a superfluous part in `shrink`, namely, the invocation of `run_rewrites` instead of simply saturating. As `shrink` finishes by

inspecting only the terms that were present at the start, removing the added terms after saturating is unnecessary. Thus, we replace `run_rewrites` with a saturation.

Lastly, we make a domain-specific improvement that does not concern Ruler's main bottleneck. As equality over $\texttt{cvec}_1$s is by nature transitive, we can group E-Classes directly when we create them based on their $\texttt{cvec}_1$. This removes the need to compare the `cvec` of every E-Class with each other for equality when creating candidate rules.

5.3 Intersecting the Saturated Classes

Now, for every example $(\vec{x_i}, y_i)$, we have found a collection of programs P_i' that satisfy it. To find a program that solves all the examples, we need to find a shared program between these E-Classes. In other words, we need to find an element in their intersection (line 10). These E-Graphs have been created within different contexts, so we require a context-sensitive intersection. For this purpose, a new algorithm (see Algorithm 3) has been invented. We will first give an intuition of the algorithm. Then, we will introduce a new kind of rewrite rule, a wildcard rule (see Definition 17), that can increase the number of programs in the intersection. Next, we will explore the divide-and-conquer algorithm as a whole that finds one program shared between all E-Classes intersected if such a program exists. As cycles form a problem for the algorithm, we will subsequently discuss how this is solved. Finally, we will dive into the complexity of the algorithm and optimizations.

5.3.1 Intersection of E-Graphs with Different Contexts

Normally, if two E-Graphs represent the same term, the corresponding E-Classes represent the same equality. Therefore, E-Graph intersection would usually boil down to the process of matching E-Classes that represent the same value and finding the terms that they all represent. In this work, however, E-Graphs get saturated sensitive to their own context. This implies that for one E-Graph, terms might be equal that are not only unequal in another, but cannot be equal; in that E-Graph, those terms interpret to non-identical values. The terms one E-Class represents might be spread over multiple E-Classes from another E-Graph that cannot be equivalent in the first. Intersection, thus, cannot simply be done by matching E-Classes. This brings us to the following major inductive insight: two E-Classes represent the same term only when (1) they represent the same E-Node, and (2) for each child of that E-node, they represent the same term.

5.3.2 Wildcard Rules

Here, we will introduce a new type of rewrite rule, the wildcard rule (see Definition 17), that can introduce holes in a program. The definition of a wildcard rule first requires the definition of a wildcard (see Definition 16).

Definition 16 (Wildcard Node)**.** A wildcard node $\square$ is a program that can be rewritten to any program p, or $\square \rightarrow p$. It can be seen as a hole or a wildcard for a program. No program can be rewritten to the $\square$ node but itself, as this would imply all programs are equivalent. Therefore, while any program can replace $\square$, it is defined to be behaviorally equivalent only to itself. ⌟

Definition 17 (Wildcard Rule)**.** A wildcard rule is a rewrite rule that contains a wildcard node $\square$ (see Definition 16). ⌟

Example 16. The following rule, where p is a metavariable, is an example of a wildcard rule:

$$p \leftrightarrow true \ ? \ p : \square$$

This rule allows for the insertion of guards into a program. Its effect will become clearer in Example 19. ⌟

5.3.3 Finding a Program in the Intersection

Here, we will explore an algorithm that is guaranteed to find a program in the intersection of the E-Classes if such a program exists. We start by finding all potentially shared E-Nodes. An E-Node is potentially shared between two E-Classes when its constant is shared and the number of arcs is equal. This separates E-Nodes that have the same constant, but have a different number of children.

Example 17. E-Graphs g and h from Figure 5.3 both have two E-Nodes with constant symbol $-$. E-Nodes g_b^1 and h_a^1 have one outgoing arc and E-Nodes g_a^1 and h_b^1 have two. Therefore, g_b^1 only gets matched with h_a^1, and g_a^1 only with h_b^1. ⌟

Thus, we take the intersection of constant symbols from each E-Class and sort the obtained intersection by the number of children each operator has (line 8). Sorting guarantees we look at constants without children first, which makes sure we find a program if the intersection is nonempty. An explanation for optimality can be found in Section 5.3.4.

For each shared constant, we try to build a program from it (line 9). If the E-Nodes compared have no children, then the constant is the program they represent (line 11). This case forms the base case of the algorithm. Else, we recursively try to find a program in the intersection of every ith child E-Class from the E-Graphs compared (line 19). Note that there might be multiple nodes in an E-Class that use the same constant. As we need to consider each potentially equal program, we loop over the cross product of these node collections (line 16). When we have found a program for every parameter, we conquer by creating the corresponding parent program; the shared constant combined with the parameters (line 21).

Example 18. Figure 5.2 shows a program in the intersection of two E-Classes, E-Class g^1 in E-Graph g with h^1 in h. We first find the corresponding E-Nodes

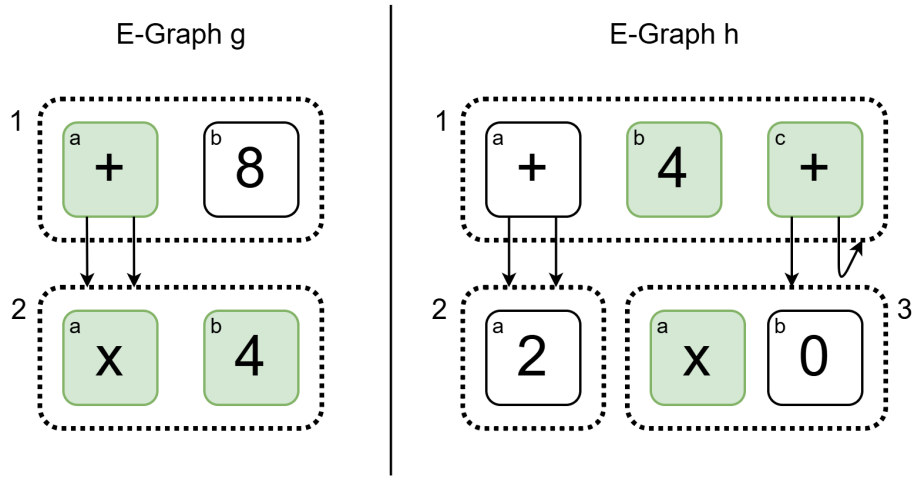

Figure 5.2: This figure shows a program, `x+4`, in the intersection of E-Class g^1 from E-Graph g with E-Class h^1 from E-Graph h in green.

by intersecting the constant symbol c together with the arc count n, represented as c_n, of each E-Node from E-Classes g^1 and h^1. When the arc count is 0, we omit it. This intersection results in: $\{8, +_2\} \cap \{4, +_2, +_2\} = \{+_2\}$. Now we know that, if the intersection is non-empty, the program must be represented by both E-Node g^1_a and h^1_a or both g^1_a and h^1_c. We first try g^1_a and h^1_a and intersect their corresponding first argument E-Class, g^2 and h^2. Their constant intersection results in $\{x, 4\} \cap \{2\} = \emptyset$. Thus, there is no program in the intersection and we return $\perp$. Therefore, the first argument of g^1_a and h^1_a cannot be filled in, proving their intersection is empty. Now we try g^1_a and h^1_c. The constant intersection of their first corresponding E-Class parameters, g^2 and h^3, results in $\{x, 4\} \cap \{x, 0\} = \{x\}$. This is an E-Node with no children and thus a complete program. We return `x`. The constant intersection of their second corresponding E-Class parameters, g^2 and h^1, results similarly in the program `4`. Now we can create a program in the intersection of g^1 and h^1, namely `x+4`. ⌟

Note that the algorithm's structure naturally extends to any number of input E-Graphs. The core logic of finding shared constants and recursively intersecting corresponding child E-Classes is applied across all provided E-Graphs simultaneously. The recursive calls are made for each corresponding child across all E-Graphs.

Wildcard rules can introduce a wildcard $\square$ to our system. Therefore, it is important that the intersection algorithm can handle them. Because $\square$ is defined to be behaviorally equivalent only to itself, it will always end up in its own E-Class. When one of the E-Classes being intersected represents $\square$, we can safely remove that E-Class from $\mathcal{C}$ (see line 3), as it inherently represents any program. If the list of intersected classes only contains one E-Class, we can simply extract a program from it (see line 7). If it were to become empty, we return $\square$ (see line 6). The fol-

Algorithm 3: INTERSECTION

Input: E-Graphs $\mathcal{G}$, E-Classes $\mathcal{C}$, initially empty global history $\mathcal{H}$, and initially empty branch history set $\mathcal{B}$
Output: A program in the intersection of $\mathcal{C}$ or $\perp$ if there is none
$\mathcal{C} \leftarrow \{c \in \mathcal{C} \mid c \text{ does not represent } \square\}$
$\mathcal{C}$ not in $\mathcal{H}$ or **return** $\mathcal{H}_{\mathcal{C}}$
$\mathcal{C}$ not in $\mathcal{B}$ or **return** $\perp$
$|\mathcal{C}| > 0$ or **return** $\square$
$|\mathcal{C}| > 1$ or **return** $\texttt{extract}(\mathcal{C})$
$C \leftarrow$ common constants of E-Classes in $\mathcal{C}$ sorted on $\#children$
for $c \in C$ **do**
    $n \leftarrow \#children$ of c
    if *n == 0* **then**
        $\mathcal{H}_{\mathcal{C}} \leftarrow c$
        return c
    else
        $\mathcal{N} \leftarrow$ for each class in $\mathcal{C}$, the E-Nodes with constant c
        for $N \in$ *all combinations over* $\mathcal{N}$ **do**
            for $i \in \{1,2,\cdots,n\}$ **do**
                $\mathcal{C}_i \leftarrow \{$the i-th argument E-Class of $u \mid u \in N\}$
                $p_i \leftarrow$ INTERSECTION$(\mathcal{G}, \mathcal{C}_i, \mathcal{B} \cup \{\mathcal{C}\}, \mathcal{H})$
            if *there is a program for each parameter of* c **then**
                $p \leftarrow$ create program $c(p_1, p_2, \cdots, p_n)$
                $\mathcal{H}_{\mathcal{C}} \leftarrow p$
                return p
$\mathcal{H}_{\mathcal{C}} \leftarrow \perp$
return $\perp$

lowing two examples serve to demonstrate the power of wildcard rules to introduce more programs.

Example 19. Take the wildcard rule discussed in Example 16

$$p \leftrightarrow true \ ? \ p : \square$$

where p is a metavariable. This rule can be used to introduce the absolute function to a domain containing only `x` and `-x`. In the context of the input `5`, we have the contextual rule

$$true \leftrightarrow x == 5 \mid x \leftarrow 5$$

Starting from program `x`, we get

$$
\begin{aligned}
x &\to true\ ?\ x : \square && (p \leftrightarrow true\ ?\ p : \square) \\
&\to x == 5\ ?\ x : \square && (true \leftrightarrow x == 5 \mid x \leftarrow 5) \\
&\to x \geq 0\ ?\ x : \square
\end{aligned}
$$

Intersecting this program with $x \geq 0\ ?\ \square : -x$, gives the absolute function. ⌟

Example 20. Depending on the domain, an interesting wildcard rule (see Definition 17) for our system is:

$$p \leftrightarrow \square\ ?\ p : p$$

In theory, applying this wildcard rule would result in the same intersection as the two general rules

$$p \leftrightarrow true\ ?\ p : p \qquad \text{and} \qquad p \leftrightarrow false\ ?\ p : p$$

because any program that can substitute the wildcard equals either $true$ or $false$. However, the specific term that should replace $true$ or $false$ resulting in a non-empty intersection might not be present. Therefore, applying the wildcard rule has a higher chance of resulting in the desired program. ⌟

5.3.4 Cycles and Optimality

There is one remaining problem, namely shared cycles. It might happen for the E-Classes being intersected that a shared child points to an E-Class that, in all cases, is already part of the term being created. In other words, we are already taking the intersection of these exact classes. In this case, the recursion would go on forever. Therefore, to potentially stop the recursion, it is necessary to detect which E-Classes have been seen together for the program being constructed in the current recursive traversal. We define a branch history $\mathcal{B}$ (see Definition 18) as a set of E-Class collections and use it to track which E-Classes have been visited together in the current recursive traversal. When going in recursion, the children get a new version of $\mathcal{B}$ that also contains $\mathcal{C}$ (see line 19). Before intersecting the E-Classes $\mathcal{C}$, we check whether $\mathcal{C} \in \mathcal{B}$ (line, 5). If this is the case, it implies we are already intersecting these classes in the current recursive traversal. Thus, if we do not return, we will repeat the steps that occurred between the first time we intersected the classes in $\mathcal{C}$ and now. Therefore, we return $\perp$ for this call instead.

Definition 18 (Branch History)**.** Let $\mathcal{G} = \{g_1, \cdots, g_n\}$ be a collection of E-Graphs. Then $\mathcal{B}$ is a set such that its members are of the form $\mathcal{C} = \{c_1, \cdots, c_n\}$ where each $c_i \in g_i$. ⌟

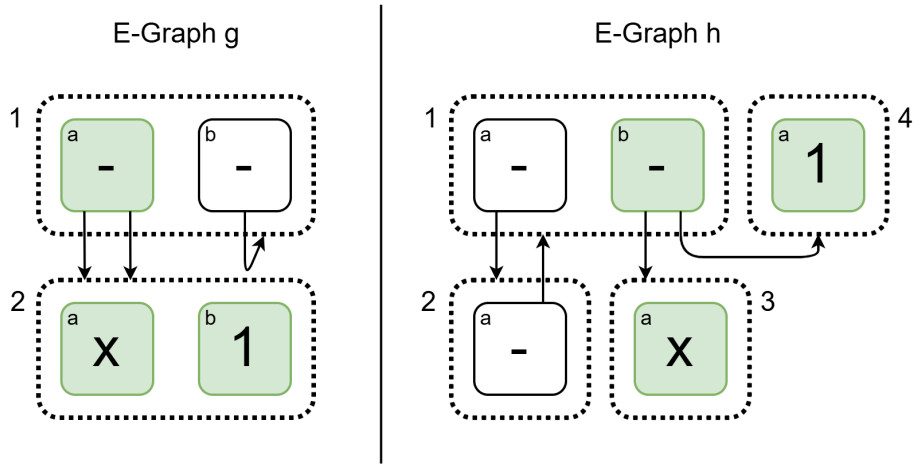

Figure 5.3: This figure shows a program, `x-1`, in the intersection of E-Class g^1 from E-Graph g with E-Class h^1 from E-Graph h in green. g^1 shares cycle $g^1_b \to g^1_b \to g^1_b$ with cycle $h^1_a \to h^2_a \to h^1_a$ in h.

Example 21. Figure 5.3 shows a program in the intersection of two E-Classes, E-Class g^1 in E-Graph g with h^1 in h. These E-Classes share a cycle. Cycle $g^1_b \to g^1_b \to g^1_b$ from g corresponds to cycle $h^1_a \to h^2_a \to h^1_a$ from h. When we naively try to find a program represented by both g^1_b and h^1_a, recursion would go on forever. The branch history $\mathcal{B}$ fixes this problem. The second time g^1 and h^1 get intersected, $\mathcal{B}$ includes $\{g^1, h^1\}$. Because at that point $\mathcal{C} \in \mathcal{B}$, the algorithm returns $\perp$. The intersection of g^1 with h^1, however, is still ongoing and will intersect g^1_a with h^1_b, resulting in program `x-1`. ⌟

Important to note is that this does not make our program non-optimal. The algorithm will still find a program iff there is a program in the intersection of $\mathcal{C}$. Assume, for a particular call where we intersect the classes in $\mathcal{C}$, we have $\mathcal{C} \in \mathcal{B}$. If this intersection were to represent a program, it would imply that said program is a larger, behaviorally equivalent version of a program in the intersection of $\mathcal{C}$. Specifically, there cannot be a constant in the intersection, as that would make it impossible to intersect $\mathcal{C}$ when $\mathcal{C} \in \mathcal{B}$. Therefore, by terminating this recursive traversal, we do not lose completeness.

5.3.5 Complexity

To analyze the worst-case complexity, we will assume there are no common cycles. Assume we intersect n classes with c common constants per class, each with m children, and k E-Nodes represented by c per class. For each common constant c, we look at all combinations over the k nodes from n classes. For each, we go in recursion for all m children up to depth d. We get:

$$T(d) = c \cdot k^n \cdot mT(d-1)$$

where d is the recursion depth and $T(1) = O(1)$. This expands to the complexity:

$$O((c \cdot k^n \cdot m)^d)$$

Thus, the most significant factor that impacts the complexity is the size of the E-Graphs. Normally, the complexity will be much lower as k will usually be at most 1, the majority of branches will terminate far earlier than max depth d, for a node we only recurse into the next child if all previous ones have successfully returned a program, and once a full program for a node is found we do not explore alternative nodes.

5.3.6 Global History

INTERSECTION uses a global history $\mathcal{H}$ that tracks the solutions found for every intersection. Every call to INTERSECTION gets a reference to $\mathcal{H}$ (see line 19). Therefore, updates to $\mathcal{H}$ are global. If a collection of classes has been fully intersected, the result gets saved in $\mathcal{H}$ (see lines 22 and 24) and returned in case they get intersected again (see line 4).

5.4 Implementation

CSES[1] is implemented in Julia [24]. The Grammar, its constraints, and the examples are represented using the Herb.jl[2] program synthesis library, which ensures inherent broad applicability across different domains. Ruler has been reimplemented in Julia as a stand-alone module using the MetaTheory.jl [25] E-Graph library for equality saturation, which has similar performance to the state-of-the-art egg [26] library. Furthermore, rewrite rules in Julia are inherently more expressive than in languages such as Rust, because they are themselves represented as native Julia expressions. This design implies that any valid Julia program can be captured and rewritten without the need for an encoding. MetaTheory.jl has likewise been used as the base for rewrite rules, E-Graphs, and equality saturation in CSES. In total, CSES consists of fewer than 1000 lines of code, making it simple and extensible.

[1] https://github.com/ViciousDoormat/CSES/tree/main/ContextSensitiveEGraphAnalysis

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

Chapter 6

Experimental Evaluation

Our experimental evaluation focuses on demonstrating the effectiveness of CSES as an example-based synthesis tool, with and without our improvements to Ruler. We separate this evaluation into the following research questions:

- **RQ1:** Does CSES perform competitively on real-world problems in relation to constrained enumeration?
- **RQ2:** Do the improvements on the bottleneck of Ruler improve efficiency?
- **RQ3:** Do the improvements on the bottleneck of Ruler help allow using a larger termset?

Lastly, we explore a case study regarding the potential effectiveness of CSES. For all experiments, we use a subset of 21 problems from SyGus SLIA (see Appendix C), which captures a wide range of real-world string manipulation tasks. Furthermore, it allows testing the performance of Ruler on more complex execution spaces than the ones it was benchmarked on. A subset is used because experimenting on the entire benchmark is not feasible due to limited time. To make our results as representative for SyGus SLIA as possible, the instances are chosen such that problems of varying difficulties are included based on [27]. For all experiments, we use a timeout of 20 minutes per problem.

All experiments were run under Windows 11 (64 bit) on an AMD Ryzen 9 7900X 12-Core Processor of 4.70 GHz, with a 32 GB RAM and a 64 MB L3 cache.

6.1 RQ1: Performance of CSES

Here, we want to compare the effectiveness of CSES with constrained enumeration. The constraints used for both enumeration and enumerating individual solutions can be found in Appendix B.1). A subset of these constraints was used to constrain the creation of the small term set by disallowing some nonsensical programs (see Appendix B.2). CSES is performed without the two optimizations for $n = 0$, $n = 1$, and $n = 2$. Execution time is used as our metric because there is no research on estimating the number of terms represented by an E-Graph.

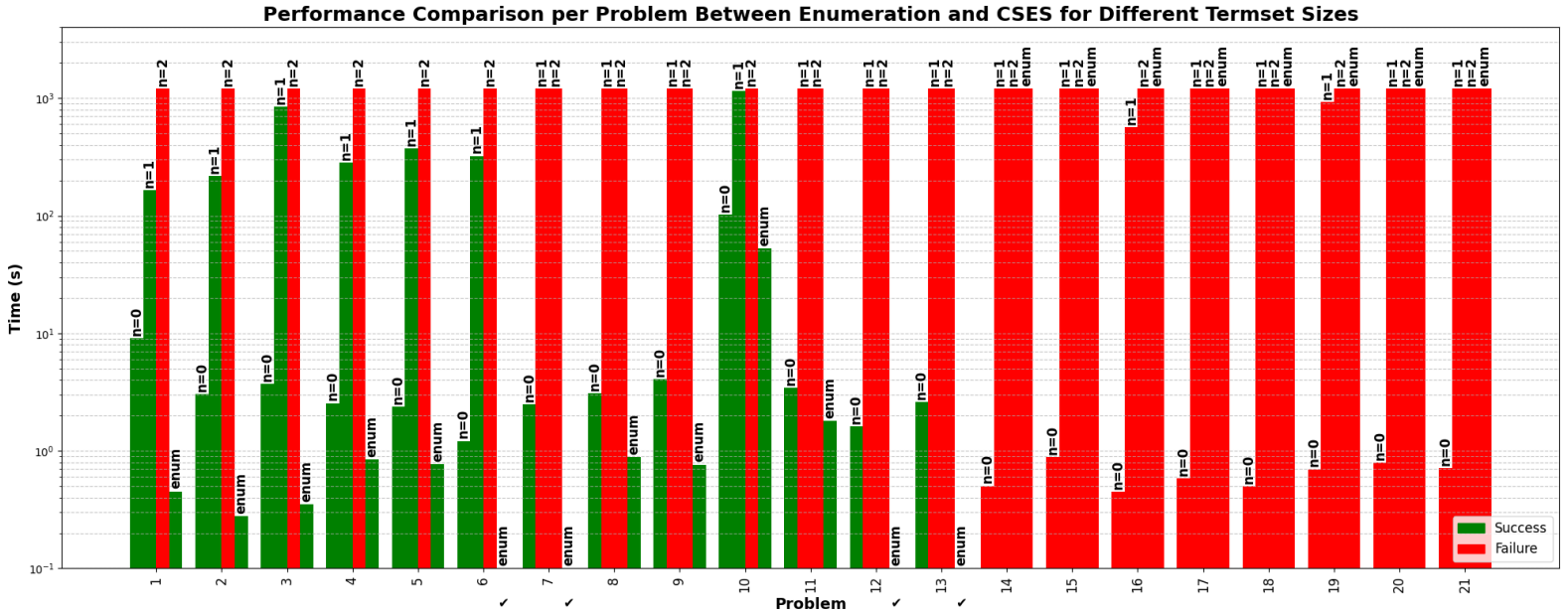

Figure 6.1: The solving time for each problem when solved by CSES without any optimizations for $n = 0$, $n = 1$, and $n = 2$, and for constrained enumeration. The color indicates whether a solution was found (green/red). A check mark replaces green in case the bar is not visible. The highest bars reached a timeout at $1200s$.

An experiment was performed to discover which problems find individual solutions within the 20-minute timeout, and whether one of those individual solutions solves all examples. The results can be found in Table 6.1. As we can see, for all problems up to and including problem 13, constrained enumeration finds an individual solution for all examples. Interestingly, for all these problems, at least one individual solution solves all examples. Problem 14 finds two individual solutions, but neither solves all examples. Lastly, the longest time needed to find all individual solutions found within the timeout is 51.57 seconds. Based on these results, we put a timeout of 1 minute on finding individual solutions in CSES.

Problem	**Found / Total**	**Universal**	**Exploring Time (s)**	**Timeout**
1	$3/3$	Y	0.06	N
2	$3/3$	Y	0.05	N
3	$3/3$	Y	0.12	N
4	$2/2$	Y	1.41	N
5	$2/2$	Y	1.43	N
6	$3/3$	Y	0.03	N
7	$4/4$	Y	0.09	N
8	$4/4$	Y	0.77	N
9	$5/5$	Y	0.87	N
10	$3/3$	Y	51.57	N
11	$2/2$	Y	2.16	N
12	$2/2$	Y	0.13	N
13	$5/5$	Y	0.17	N
14	$2/3$	N	0.44	Y
15	$0/4$	N	$\perp$	Y
16	$0/2$	N	$\perp$	Y
17	$0/4$	N	$\perp$	Y
18	$0/3$	N	$\perp$	Y
19	$0/3$	N	$\perp$	Y
20	$0/3$	N	$\perp$	Y
21	$0/4$	N	$\perp$	Y

Table 6.1: For each problem, the number of individual solutions found with respect to the total number of examples, whether one of the individual solutions is universal and thus solves all examples (Y/N), the time required to find all individual solutions that were found, and whether the timeout of 20 minutes was reached (Y/N). If no individual solution was found before the timeout, we use $\perp$. The individual solutions are found with constrained enumeration. Note that our aim is not to solve the problem, but instead to find individual solutions.

The results of our experiment are shown in Figure 6.1. Both Constrained enumeration and CSES with $n = 0$ find a solution for problems 1 up to and including 14. Constrained enumeration is always fastest when a solution is found. However, CSES with $n = 0$ always finishes its execution, whereas constrained enumeration times out for all problems starting from 15. CSES with $n = 1$ times out for problems 7, 8, and 9 and all problems starting from 11. It did find a solution to the problems when it did not time out. Lastly, CSES with $n = 2$ always times out.

First of all, we observe that individual solutions are almost always either too hard to find, or already a universal solution. Thus, for our subset of SyGus SLIA, finding individual solutions through enumeration does not seem significantly easier than finding a universal solution.

Prob	**Avg #C**		**% Diff**	**Time (s)**		**Found / Total**	
	With R	No R		With R	No R	With R	No R
1	317.00	459.00	30.94	151.74	164.54	3/3	3/3
2	422.33	549.33	23.12	212.82	218.14	3/3	3/3
3	906.00	1213.00	25.31	474.59	841.65	3/3	3/3
4	601.00	767.00	21.64	202.94	281.67	2/2	2/2
5	601.00	767.00	21.64	273.79	371.59	2/2	2/2
6	389.67	510.00	23.59	206.13	322.49	3/3	3/3
7	762.50	992.50	23.17	770.36	$\perp$	4/4	4/4
8	668.00	851.00	21.50	862.87	$\perp$	4/4	4/4
9	1059.40	1430.00	25.92	$\perp$	$\perp$	5/5	2/5
10	674.00	783.33	13.96	724.19	1144.98	3/3	3/3
11	1102.50	1435.50	23.00	790.45	$\perp$	2/2	2/2
12	1072.00	1411.00	24.03	1029.11	$\perp$	2/2	2/2
13	4374.00	4050.00	-8.00	$\perp$	$\perp$	2/5	1/5
14	3429.00	4181.00	17.99	$\perp$	$\perp$	1/3	1/3
15	1027.67	1170.00	12.16	$\perp$	$\perp$	3/4	2/4
16	373.00	490.00	23.88	361.03	569.28	2/2	2/2
17	541.00	685.00	21.02	$\perp$	$\perp$	4/4	3/4
18	600.00	766.00	21.67	953.78	$\perp$	3/3	3/3
19	373.00	490.00	23.88	629.17	926.39	3/3	3/3
20	3502.00	3797.00	7.77	$\perp$	$\perp$	1/3	1/3
21	556.00	718.00	22.56	$\perp$	$\perp$	3/4	3/4
Avg	1112	1310	20.04	359.60*	537.86	–	–

Table 6.2: For each problem, the average number of candidate rules C found per example with and without general rules R, together with the percental difference ($\frac{\text{No R}-\text{With R}}{\text{No R}} \cdot 100$). The time columns show the time Ruler took to run with and without R. The last two columns show the number of examples where candidate rules were found within the timeout out of the total number of examples. The symbol $\perp$ indicates no candidate rules were found within the timeout.

Furthermore, we can see that, for any value of n, CSES only finds a solution for problems for which a universal solution is found as one of the individual solutions. When a universal solution is part of the termset, CSES effectively functions as an overhead over enumeration, as every example solution space will contain said solution, making finding the solution trivial. This explains CSES for $n = 0$ finds a solution only when enumeration does too. The same might be true for $n = 1$, but we cannot currently guarantee this for problems where it times out. When we set n to 2, we see that our system always times out, indicating Ruler becomes too large of a bottleneck.

6.2 RQ2: Ruler Improvements

Here, we want to measure the effectiveness of the two main improvements to Ruler, the general ruleset and the new rule selection approach. The other improvements are more obvious in their effect; thus for the sake of time, we do not measure their effect.

First, we run CSES for $n = 1$ on our set of problems both with and without a general ruleset R (see Appendix A) containing 24 rules. For each problem, Ruler's execution time was measured, together with the average number of candidate rules found over the examples. If Ruler times out, the average is taken only over the found sizes of the generated candidate rulesets. Thus, this indicates the advantage gained from the average decrease in candidate rules. The results of this experiment can be found in Table 6.2.

We see that, for each problem except 13, less candidate rules are found when R is used. The biggest difference is obtained for problem 14 with 752 fewer average candidate rules, and the percental improvement is generally around 20%. We also see that every problem that finishes execution is faster when R is used. Lastly, 5 more problems finish executing.

Second, we run CSES for $n = 1$ on our set of problems with the standard and new selection approach. For each problem, the average execution time of the rule selection procedure was measured. If Ruler times out, the average is taken only over the finished instances. Thus, this represents the average advantage gained. As the main thing affected by this change is the selection time, no other measurements were taken. The results of this experiment can be found in Table 6.3.

We see that only problems 9, 11, and 15 have a faster average execution time for the new select procedures. All other problems have a lower average rule selection execution time. Furthermore, nine problems finish rule selection for less examples, of which two do not finish one procedure at all.

As expected, using R generally decreases the number of candidate rules, resulting in a faster execution time. The increase in average candidate rules for problem 13 can be explained because, without R, CSES is unable to create the candidate set for the second example in time, whereas it does create it when using R. Thus,

*Average taken only over problems with found candidate rules within the timeout without R.

for the second example, probably a larger candidate set is generated, resulting in a higher average. However, the new selection procedure does not seem to be an improvement. Most problems get a worse execution time. Thus, `shrink`ing less often but with a larger ruleset does not generally improve performance. Why specifically three problems do have a better execution time is not clear.

Problem	**Avg Selection Time (s)**		**% Diff**	**Fin. / Total**	
	New	**Old**		**New**	**Old**
1	65.33	42.23	-54.70	3/3	3/3
2	96.92	57.20	-69.44	3/3	3/3
3	345.05	242.23	-42.45	3/3	3/3
4	216.42	110.77	-95.38	2/2	2/2
5	279.25	149.42	-86.89	2/2	2/2
6	224.78	82.20	-173.45	3/3	3/3
7	502.84	238.83	-110.54	2/4	3/4
8	486.01	264.94	-83.44	2/4	3/4
9	479.54	544.20	11.88	1/5	1/5
10	390.13	262.57	-48.58	2/3	3/3
11	543.22	606.81	10.48	1/2	1/2
12	$\perp$	904.70	$-\infty$	0/2	1/2
13	$\perp$	$\perp$	$\perp$	0/5	0/5
14	$\perp$	$\perp$	$\perp$	0/3	0/3
15	630.77	664.43	5.07	1/4	1/4
16	535.53	209.44	-155.70	1/2	2/2
17	440.75	367.37	-19.97	1/4	2/4
18	638.55	418.29	-52.66	1/3	2/3
19	518.18	228.35	-126.92	1/3	3/3
20	$\perp$	$\perp$	$\perp$	0/3	0/3
21	$\perp$	428.96	$-\infty$	0/4	2/4
Avg	399.58	280.58*	-68.29	–	–

Table 6.3: For each problem, the average rule selection time over the examples for which `choose_eq` finished execution within the timeout, with and without the new select approach, including the percental difference ($\frac{\text{Old}-\text{New}}{\text{New}} \cdot 100$). and the number of examples `choose_eq` finished (Fin.) out of the total number.

6.3 RQ3: Performance of Improved CSES

Here, we examine whether our optimizations allow solving more problems. We have seen general rules provide a significant improvement, and the new rule selec-

*Average taken only over problems where `choose_eq` New finished execution.

tion approach sometimes has a positive effect. To this end, we perform CSES on all problems for $n = 1$ and $n = 2$ once with only general rules and once with both general rules and the new selection approach. The results are shown in Figure 6.4.

Our results show that both approaches do solve more problems and finish execution more often than CSES without any improvements for $n = 1$. All additionally solved problems are also solved by enumeration. Thus, CSES still only solves problems for which there is an individual solution. Furthermore, we see only a small difference in performance between our two approaches. Lastly, CSES still times out for all problems with $n = 2$.

It seems that, for $n = 1$, without individual solutions, the termset does not contain the terms required to find a sufficiently expressive rewrite ruleset to rewrite an individual solution to a universal one. Therefore, these problems cannot be solved. Furthermore, our optimizations do not have a large enough effect to make CSES with $n = 2$ finish execution in time. The bottleneck of Ruler has too much of an impact. This suggest that either a faster ruleset generation mechanism is required that has sufficient performance, or a technique that allows having the terms that form crucial rewrite rules in our termset without enumerating exponentially more programs.

6.4 Case Study

We will look at two problems that show the potential power of CSES. These problems show that even with a small n, contextual rewrite rules can exist that express contextual equivalences able to generate a solution program. If the grammar allows these to be found fast, potentially together with a set of wildcard rules, the inefficiency of Ruler might not be a problem.

6.4.1 Powerful Contextual Rewrite Rules

```
ntString = " " | arg1
ntString = substr(ntString, ntInt, ntInt)
ntInt = 1
ntInt = ntInt + ntInt
ntInt = len(ntString)
ntInt = indexof(ntString, ntString, ntInt)
```

Figure 6.2: A grammar snippet with the necessary parts of the grammar from problem 19 `get_last_name_from_name`.

Take the grammar from Figure 6.2 and problem `get_last_name_from_name` from RQ1:

```
[[(arg1 => "Park Kim")], "Kim"),
 [(arg1 => "Lee Kim")],  "Kim"),
 [(arg1 => "Kim Lee")],  "Lee")]
```

Because of the nature of the grammar, this problem requires a solution of a size of at least 3 nested programs. An example of such a solution is:

```
substr(arg1, indexof(arg1, " ", 1)+1, len(arg1))
```

It contains a call to `indexof` inside of a summation contained in a call to `substr`. Constrained enumeration will not find this solution in time. Let's zoom in.

Note the term `indexof(arg1, " ", 1)`. The corresponding $\texttt{cvec}_1$s for our examples would be 5, 4, and 4 respectively. We will not find terms with equal $\texttt{cvec}_1$s if we use an n lower than 3. For example, 4 can be constructed as $1 + 1 + 1 + 1$, which requires $n = 3$. The same holds true for `len(arg1)`, which equals 8, 7, and 7 respectively.

Adding the output to the grammar at the start allows us to find more of such crucial terms. If we add the characteristic vectors found to the grammar and start over, we will find rewrite rules that prove these contextual equivalences.

Furthermore, we will find terms that solve individual examples. These are all the terms and rules required to represent, for each example, our solution program. Therefore, with this altered grammar, CSES solves this problem already with $n = 1$, whereas enumeration will only take longer as the grammar size has increased.

Example 22. Take the first example. As we add `Kim` to the grammar at the start, we will enumerate the term `indexof("Kim", arg1, 1)`, allowing us to find `6` alongside the earlier mentioned $\texttt{cvec}_1$s. Thus, we will enumerate the individual solution `substr(arg1, 6, 8)` when we re-enumerate, and find the contextual equivalence between `5` and `indexof(arg1, " ", 1)`, `6` and `5+1`, and `8` and `len(arg1)`. Thus, the E-Graph that will represent the solution space for this example will contain our program solution after being saturated with these equivalences. ⌟

6.4.2 More Possible with Wildcard Rules

Take the grammar from Figure 6.3 and problem `split_text_string_at_specific_character` from RQ1:

```
[[(arg1 => "011016_assignment.xlsx", arg2 => 1)],
   "011016"),
 [(arg1 => "011016_assignment.xlsx", arg2 => 2)],
   "assignment.xlsx"),
 [(arg1 => "030116_cost.xlsx", arg2 => 1)],
   "030116"),
 [(arg1 => "030116_cost.xlsx", arg2 => 2)],
   "cost.xlsx")]
```

```
ntString = "_" | arg1
ntString = ntBool ? ntString : ntString
ntString = substr(ntString, ntInt, ntInt)
ntInt = 1 | arg2
ntInt = ntInt + ntInt | ntInt - ntInt
ntInt = len(ntString)
ntInt = indexof(ntString, ntString, ntInt)
ntBool = ntInt == ntInt
```

Figure 6.3: A grammar snippet with the necessary parts of the grammar from SyGus problem `split_text_string_at_specific_character`.

A solution to this problem expressible by the grammar is, for example:

```
arg2 == 1 ?
substr(arg1, 1, indexof(arg1, "_", 1)-1) :
substr(arg1, 1, indexof(arg1, "_", 1)+1)
```

Similar to before, this solution will not be found in time by enumeration. It would even take a lot longer to find this program than the solution to the previous problem. If we again add the values of `indexof(arg1, "_", 1)` for each example to the grammar, CSES will find the `then` case as a solution for examples one and three, and the `else` case for examples two and four. Therefore, the solution program will be found for each example with the wildcard rules

$$p \leftrightarrow true\ ?\ p : \square \qquad \text{and} \qquad p \leftrightarrow false\ ?\ \square : p$$

Thus, with this altered grammar and using these wildcard rules, CSES again solves this problem, whilst enumeration does not.

6.4.3 Reflection

As we have seen, CSES often fails to find a solution program because key contextual rewrite rules are not discovered. Said rewrite rules cannot be discovered because the grammar lacks the expressiveness required to represent the key terms needed to form these rules at a low value of n. Therefore, adding such terms to the grammar makes it possible for CSES to solve such problems. However, enlarging the grammar too aggressively might result in a termset that is too large for the system to handle efficiently. Interestingly, in both case studies, the crucial terms were generated by our system as $\texttt{cvec}_1$s of programs applied to a variable, after the example output was added to the grammar. This suggests a potential strategy: add all unique $\texttt{cvec}_1$ values to the grammar from $n = 1$ programs containing a variable.

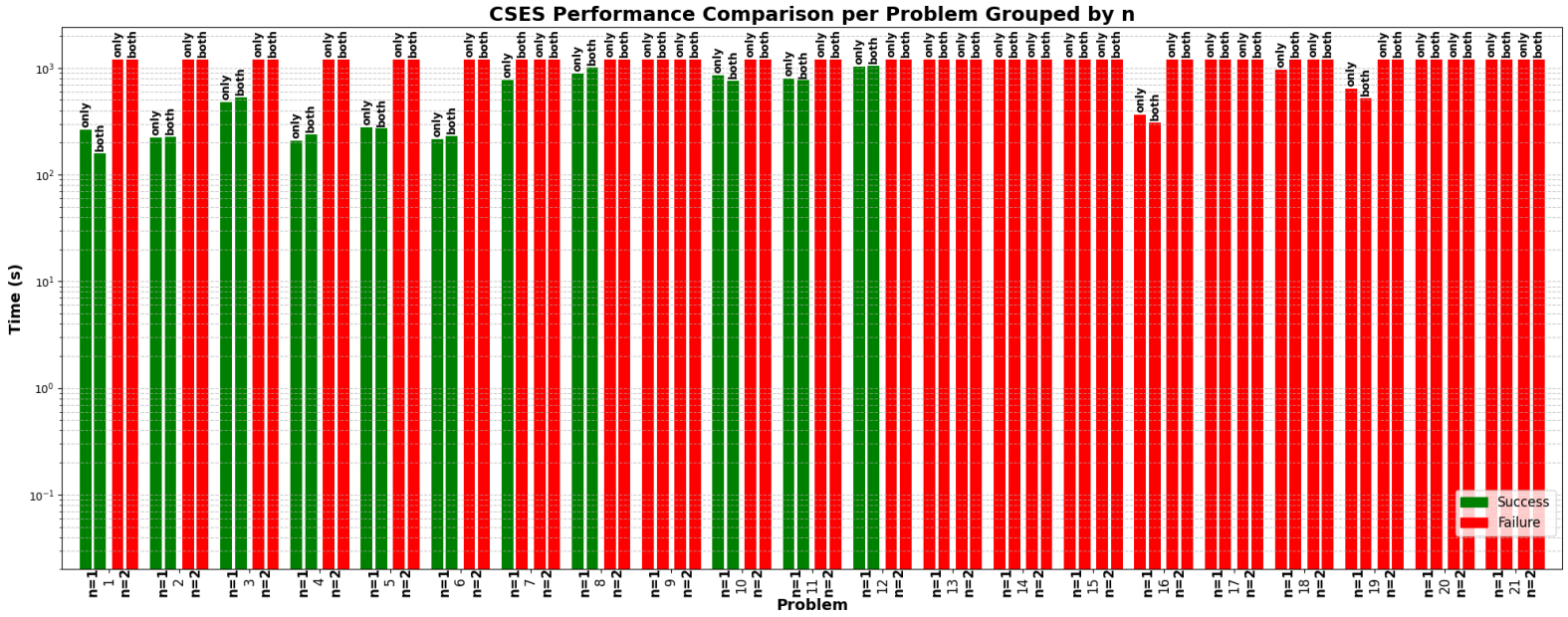

Figure 6.4: The solving time for each problem when solved by CSES with general rules **only** and general rules and the new select approach **both**, for $n = 1$ and $n = 2$. The color indicates whether a solution was found (green/red). The highest bars reached a timeout at $1200s$.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We set out to explore how the search space of example-based synthesis could be reduced by focusing only on programs that solve at least one IO example. To this end, we introduced Context-Sensitive E-Graph Saturation, a novel method that constructs and leverages contextual equivalences to prune the search space. By doing so, we aimed to address the our central research question: How can we limit the search space to programs that behave similarly to the one desired?

We compared CSES to constrained enumeration on problems from SyGus SLIA and found that CSES only successfully solves problems enumeration solves too, as the termset contains a universal solution. However, enumeration consistently outperformed CSES in terms of solving speed. Importantly, CSES was able to complete execution in cases where enumeration timed out. We also evaluated the improvements of using a general ruleset and the new rule selection approach. We found that a general ruleset substantially decreased the number of candidate rules and reduced execution time of Ruler. In contrast, the new rule selection approach did not yield consistent benefits and. Last, we examined whether these optimizations allow solving more problems. CSES was able to finish execution for more problems and solve additional ones compared to the unoptimized baseline. However, CSES still only solved problems for which individual solutions were already present, and it still consistently failed with $n = 2$. This indicates that, while our optimizations improved performance, they are insufficient to overcome the bottleneck posed by Ruler when dealing with larger termsets. These results also revealed a fundamental limitation: without key contextual rewrite rules, CSES cannot go beyond what enumeration achieves, highlighting the importance of termset expressiveness.

Future research aimed at faster ruleset generation and creating a more expressive ruleset could enable CSES to go beyond enumeration and realize its full potential.

7.2 Future Work

As we have seen, individual solutions have the potential to effectively help find solution programs. However, in our experiments, individual solutions were almost always either not found or already a universal solution. A better approach to finding individual solutions could thus greatly improve the system. This approach could better use the fact that the solution targets only one example and could be based on [28]. Another way to tackle this problem would be the approach discussed in the case study (see Subsection 6.4.3), or other methods to alter the initial grammar. Similarly, it would be interesting to test the system on other domains with more easily discoverable individual solutions, where a standard solution is not often the universal solution.

Another way to improve upon the method would be to improve upon Ruler. As we have seen, both Enumo and Isaria perform equality saturation in phases with a domain-specific approach. Similarly, a domain-specific selection approach for this system might allow to make the rule selection part of Ruler significantly more efficient. A possible heuristic could involve constructing a bipartite graph that maps (sub)terms to E-Classes and using a matching to guide rule selection. Other worthwhile improvements might be more general. An example is the insight that, with Ruler, rules are significantly less likely to get trimmed the higher their heuristic score is, as we always go through the candidate set in the same order. We `shrink` the last rules using the first ones, but never the other way around. Differing in order might be more efficient. Furthermore, as we have only analyzed the performance impact of two improvements, further research is required regarding the impact of the other discussed optimizations. Lastly, it might be better to use another system entirely to find contextual rewrite rules, which could be made specifically for this domain.

Other valuable research would be to explore an extension of this method to allow higher-order functions. A function that returns a function would require the value of a `cvec` to be a function. However, this raises the challenge that we would like to compare such `cvecs` not by syntax but by behavior, potentially requiring `cvecs` containing `cvecs`. Likewise, the most straightforward way to allow lambda functions as parameters would be to instantiate every lambda. However, this would remove the generality we desired by allowing them; we can only discover programs where the lambda is already applied. For example, we might find `map (+1) arg1`, but not the more general `map` function itself. One possible solution is to require one of the example arguments to be the lambda. Another is to treat the lambda as part of the output of `map`, producing a `cvec` of unfinished calls where the lambda still needs to be applied.

Similarly, allowing recursive programs would make the system more powerful. Basic E-Graphs are, by their nature, unable to represent recursive programs. An altered version that can represent them could create many new possibilities within this domain and many others where E-Graphs can be applied.

Lastly, the performance of the intersection algorithm on large E-Graphs has not

been analyzed. Therefore, no statements about its performance can be drawn beyond the shown complexity. This choice was made as the only existing method to create large contextual E-Graphs is CSES, which is simply too slow to create many large-scale E-Graphs in the available time. More research thus needs to be performed to benchmark this algorithm. Other interesting research in this regard would be a (heuristic) approach to measure the number of terms represented by an E-Graph, which would allow for a better comparison between CSES and other systems like enumeration.

Bibliography

[1] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[2] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[3] M. Sun, A. Lo, M. Guo, J. Chen, C. W. Coley, and W. Matusik, "Procedural synthesis of synthesizable molecules," in *The Thirteenth International Conference on Learning Representations*, 2025.

[4] D. Maletskyi and V. Shymanskyi, "Genome compression using program synthesis." in *IDDM*, 2023, pp. 310–321.

[5] S. N. Guria, J. S. Foster, and D. Van Horn, "Absynthe: Abstract interpretation-guided synthesis," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1584–1607, 2023.

[6] S. Barke, H. Peleg, and N. Polikarpova, "Just-in-time learning for bottom-up enumerative synthesis," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.

[7] D. Sannella and A. Tarlecki, "On observational equivalence and algebraic specification," *Journal of Computer and System Sciences*, vol. 34, no. 2-3, pp. 150–178, 1987.

[8] L. Brodo, R. Bruni, and M. Falaschi, "Sos rules for equivalences of reaction systems," in *International Workshop on Functional and Constraint Logic Programming*. Springer, 2020, pp. 3–21.

[9] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, "Rewrite rule inference using equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–28, 2021.

[10] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 215–225, 1975.

[11] A. P. Ershov, "On programming of arithmetic operations," *Communications of the ACM*, vol. 1, no. 8, pp. 3–6, 1958.

[12] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 264–276.

[13] D. Suciu, Y. R. Wang, and Y. Zhang, "Semantic foundations of equality saturation," *arXiv preprint arXiv:2501.02413*, 2025.

[14] S. Thomas and J. Bornholt, "Automatic generation of vectorizing compilers for customizable digital signal processors," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 19–34.

[15] A. Pal, B. Saiki, R. Tjoa, C. Richey, A. Zhu, O. Flatt, M. Willsey, Z. Tatlock, and C. Nandi, "Equality saturation theory exploration á la carte," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 1034–1062, 2023.

[16] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 422–436, 2017.

[17] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 420–435, 2018.

[18] X. Wang, I. Dillig, and R. Singh, "Program synthesis using abstraction refinement," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2017.

[19] S. Coward, G. A. Constantinides, and T. Drane, "Abstract interpretation on e-graphs," *arXiv preprint arXiv:2203.09191*, 2022.

[20] X. Wang, G. Anderson, I. Dillig, and K. L. McMillan, "Learning abstractions for program synthesis," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 407–426.

[21] P. S. Sarker, "Macro-actions for pddl: A dynamic approach," 2025.

[22] Y. Yoon, W. Lee, and K. Yi, "Inductive program synthesis via iterative forward-backward abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1657–1681, 2023.

[23] J. de Jong, "Speeding up program synthesis using specification discovery," 2023.

[24] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[25] A. Cheli, "Metatheory.jl: Fast and elegant algebraic computation in julia with extensible equality saturation," *Journal of Open Source Software*, vol. 6, no. 59, p. 3078, 2021.

[26] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, 2021.

[27] T. Hinnerichs, B. Swinkels, J. de Jong, R. G. Reid, T. Magirescu, N. Yorke-Smith, and S. Dumancic, "Modelling program spaces in program synthesis with constraints," *arXiv preprint arXiv:2508.00005*, 2025.

[28] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 2017, pp. 319–336.

Appendix A

General Rewrite Rules

Below is a list of general rewrite rules used in the experiments.

$$\texttt{concat}(a,\ \texttt{""}) \rightarrow a$$
$$\texttt{concat}(\texttt{""},\ a) \rightarrow a$$
$$\texttt{replace}(a,\ b,\ b) \rightarrow a$$
$$\texttt{replace}(\texttt{""},\ a,\ b) \rightarrow \texttt{""}$$
$$\texttt{toint}(\texttt{tostr}(a)) \rightarrow a$$
$$\texttt{tostr}(\texttt{toint}(a)) \rightarrow a$$
$$a + 0 \leftrightarrow a$$
$$0 + a \leftrightarrow a$$
$$a - 0 \leftrightarrow a$$
$$a == a \rightarrow \mathit{true}$$
$$\texttt{prefixof}(a,\ a) \rightarrow \mathit{true}$$
$$\texttt{prefixof}(\texttt{""},\ a) \rightarrow \mathit{true}$$
$$\texttt{suffixof}(a,\ a) \rightarrow \mathit{true}$$
$$\texttt{suffixof}(\texttt{""},\ a) \rightarrow \mathit{true}$$
$$\texttt{contains}(a,\ a) \rightarrow \mathit{true}$$
$$\texttt{contains}(a,\ \texttt{""}) \rightarrow \mathit{true}$$
$$\texttt{concat}(a,\ \texttt{concat}(b,\ c)) \leftrightarrow \texttt{concat}(\texttt{concat}(a,\ b),\ c)$$
$$\texttt{len}(\texttt{concat}(a,\ b)) \leftrightarrow \texttt{len}(a) + \texttt{len}(b)$$
$$\texttt{substr}(a,\ 1,\ \texttt{len}(a)) \rightarrow a$$
$$\texttt{substr}(a,\ 1,\ 0) \rightarrow \texttt{""}$$
$$\texttt{prefixof}(a,\ \texttt{concat}(a,\ b)) \rightarrow \mathit{true}$$
$$\texttt{suffixof}(a,\ \texttt{concat}(b,\ a)) \rightarrow \mathit{true}$$
$$\texttt{contains}(\texttt{concat}(a,\ b),\ a) \rightarrow \mathit{true}$$
$$\texttt{contains}(\texttt{concat}(b,\ a),\ a) \rightarrow \mathit{true}$$

Appendix B

Constraints

Here, we show a list of constraints used for our experiments implemented in Herb.jl. Expressions `A` and `B` represent metavariables. We use a collection of forbidden, unique, and ordered constraints. A detailed explanation of the constraints can be found in [27].

B.1 Constraints for Enumeration

The following constraints are used for enumeration, both to find the entire solution and as part of CSES to find individual solutions. A forbidden constraint constrains the creation of particular (sub)programs, making them not allowed. The following forbidden constraints are used:

```
forbidden(concat(A,""))
forbidden(concat("",A))
forbidden(replace(A,B,B))
forbidden(to_str(to_int(A)))
forbidden(to_int(to_string(A)))
forbidden(if A then B else B)
forbidden(if true then A else B)
forbidden(if false then A else B)
forbidden(A+0)
forbidden(0+A)
forbidden(A-0)
forbidden(A==A)
forbidden(prefixof(A,A))
forbidden(prefixof("",A))
forbidden(suffixof(A,A))
forbidden(suffixof("",A))
forbidden(contains(A,A))
forbidden(contains(A,""))
forbidden(len(to_str(A::Int)))
```

Aside from these forbidden constraints, we also forbid the creation of particular expressions over constant digits and strings. For the following list, `n` and `m` represent a constant digit and `s` a constant string:

```
forbidden(n == m)
forbidden(len(s))
forbidden(replace(s,A,B))
forbidden(substr(s,A,B))
forbidden(prefixof(A,s))
forbidden(suffixof(A,s))
forbidden(contains(s,A))
forbidden(indexof(s,A,B))
```

A unique constraint constrains the times a node can appear, forcing it to appear at most once. The following unique constraints are used:

```
unique(contains)
unique(suffixof)
unique(prefixof)
```

An ordered constraint constrains the order of metavariables. A term can only be made if the metavariables are in sorted order. The following ones are used:

```
ordered(A == B)
```

B.2 Constraints for Small Terms

The following constraints were used in CSES to remove nonsensical programs from the domain. For the following list, and `s` represents a constant string:

```
forbidden(len(s))
forbidden(replace(s,A,B))
forbidden(substr(s,A,B))
forbidden(prefixof(A,s))
forbidden(suffixof(A,s))
forbidden(contains(s,A))
forbidden(indexof(s,A,B))
```

Appendix C

SyGus Problem Subset

The set of programs from SyGus used for the experiments in this thesis.

SyGuS Problem	Assigned Problem Number
problem_convert_numbers_to_text	1
problem_convert_text_to_numbers	2
problem_cell_contains_specific_text	3
problem_replace_one_character_with_another	4
problem_remove_text_by_matching	5
problem_count_total_characters_in_a_cell	6
problem_compare_two_strings	7
problem_change_negative_numbers_to_positive	8
problem_remove_unwanted_characters	9
problem_remove_characters_from_left	10
problem_join_first_and_last_name	11
problem_37281007	12
problem_19558979	13
problem_clean_and_reformat_telephone_numbers	14
problem_44789427	15
stackoverflow9	16
remove_file_extension_from_filename	17
get_last_name_from_name_with_comma	18
problem_get_last_name_from_name	19
problem_12948338	20
problem_30732554	21

Table C.1: A subset of problems from the SyGus SLIA benchmark[1].

[1] https://github.com/Herb-AI/HerbBenchmarks.jl/tree/master/src/data/SyGuS/PBE_SLIA_Track_2019