# A Procedure for SyGuS Solution Fitting via Matching and Rewrite Rule Discovery

Abdalrhman Mohamed[†], Andrew Reynolds[*], Clark Barrett[†], and Cesare Tinelli[*]
[†]Stanford University, Stanford, CA, USA, ✉ abdal@stanford.edu [*]The University of Iowa, Iowa City, IA, USA

***Abstract*—Syntax-guided synthesis (SyGuS) is a recent software synthesis paradigm in which an automated synthesis tool is asked to synthesize a term that satisfies both a semantic and a syntactic specification. We consider a special case of the SyGuS problem, where a term is already known to satisfy the semantic specification but may not satisfy the syntactic one. The goal is then to find an equivalent term that additionally satisfies the syntactic specification, provided by a context-free grammar. We introduce a novel procedure for solving this problem which leverages pattern matching and automated discovery of rewrite rules. We also provide an implementation of the procedure by modifying the SyGuS solver embedded in the CVC5 SMT solver. Our evaluation shows that our new procedure significantly outperforms the state of the art on a large set of SyGuS problems for standard SMT-LIB theories such as bit-vectors, arithmetic, and strings.**

## I. Introduction

Program synthesis is a powerful technique with many potential applications, including program optimization, loop invariance generation, and protocol synthesis [1], [2], [3], [4]. Syntax-guided Synthesis [5] (SyGuS) is a particular paradigm for program synthesis in which the goal is to generate correct functional code from a high-level description of the desired program behavior. This high-level description is typically represented as a set of *semantic constraints*, logical formulas expressing the properties the program should satisfy, and *syntactic constraints*, which dictate the structure and syntax of the program and are commonly encoded as a formal grammar.

Program synthesis is generally undecidable and is considered to be a challenging task, even in restricted settings. Nevertheless, in the past decade, several efficient SyGuS solvers have emerged [6], [7], [8], [9], [10]. These solvers are typically *enumerative*, with a few notable exceptions [6], [10]. In an enumerative approach to syntax-guided synthesis, the solver uses the provided grammar to generate a list of terms that meet the syntactic constraints. These terms are then passed to a backend reasoner, often an SMT solver, to determine whether or not they meet the semantic constraints as well.

We consider a special case of the SyGuS problem in which the semantic specification for the function $f(\vec{x})$ to synthesize is already known to be satisfied by a given solution term $t[\vec{x}]$ with free variables $\vec{x}$. The problem, which we call the *SyGuS solution fitting problem*, is to synthesize from $t$ and a given grammar $G$ a term $t'$ that is equivalent to $t$ and is generated by $G$. It can also be seen an instance of the SyGuS problem, where the semantic specification is the formula $\forall \vec{x}.\, f(\vec{x}) \approx t[\vec{x}]$. Procedures for solving this problem can be understood as refinement procedures, where additional syntactic restrictions are imposed.

We propose a procedure for the SyGuS solution fitting problem that leverages both matching and rewrite rule synthesis to find a term in the language of $G$ that is equivalent to the input term $t$. At a high level, the procedure matches $t$ with terms generated by the production rules of the input grammar, thereby generating a set $S$ of smaller terms to synthesize. We then augment $S$ by using rewrite rule synthesis to find new terms that are equivalent to those in $S$. Finally, we use enumerative SyGuS to find derivations for a subset of $S$ that is sufficient to construct a term equivalent to $t$.

One possible use case for our procedure is when a user has successfully solved a SyGuS conjecture for a (complete) semantic specification $\varphi$ and a grammar $G$, and then needs a solution in the language of a revised grammar $G'$, which perhaps includes desirable and more stringent syntactic requirements. If the previous solution term $t$ does not fit the updated grammar, our procedure can be used to construct from $t$ an equivalent term that does. A second use case occurs within certain approaches for solving SyGuS problems [10] that focus solely on satisfying the semantic component of specifications. Our procedure can then be used to impose a posteriori syntactic constraints on the solution.

*Contributions:* We propose a novel approach for the SyGuS solution fitting problem. Our contributions include:

- A new procedure for this problem which combines matching, dynamic rewrite rule discovery, and enumerative SyGuS, and is parametric in the background theory of the semantic constraints.
- An implementation of this approach in the SyGuS solver of CVC5.
- A detailed evaluation of the procedure showing significant performance improvements over other SyGuS solvers on a set of crafted benchmarks, as well as a set of mutated benchmarks from the standard SyGuS library. Notably, the procedure scales well across multiple standard SMT-LIB theories, including bit-vectors, arithmetic, and strings.

We present related approaches and preliminaries in the rest of this section. Our approach and contributions are detailed in Section II. We evaluate our approach against other methods in Section III and conclude by outlining potential future directions in Section IV.

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_27

This article is licensed under a Creative Commons Attribution 4.0 International License

### A. Related Work

Typical methods for solving SyGuS problems are enumerative in nature. However, some works have explored divide-and-conquer methods to solve restricted classes of SyGuS problems. For example, the SyGuS solver **STUN** [11] divides the input space of the function $f$ to synthesize into subsets, enumerates expressions that are correct in these subsets, and then combines them using a *unification* operator (e.g., the if-then-else operator). **STUN** is fairly effective in solving general SyGuS problems. However, it requires domain knowledge to identify suitable unification operators and efficient program generation algorithms. Moreover, the desired unification operator may not be available in the provided grammar.

**EUSolver** [6] adopts a similar approach to **STUN** by using the ite operator and selected predicates to define the target function by cases, reformulating the SyGuS problem as a decision tree learning problem. This approach led to successful results in the SyGuS solver competition in 2016 and 2017 [12], [13]. A major limitation is that **EUSolver** requires point-wise specifications of the semantic constraints (relating an input point to its output, but not the outputs of different inputs) and a grammar that both contains the ite operator and can be decomposed into a term grammar and a predicate grammar.

Other work [10], implemented in the CVC4 SMT solver, addresses a class of synthesis problems called *single-invocation* problems, where the occurrences of the target function in the semantic constraints are all applied to the same input tuple. For these problems, CVC4 first looks for a solution satisfying just the semantic constraints and then tries to find an equivalent term within the grammar using an ad-hoc procedure based on matching against the grammar rules of the syntactic specification ( [10], Section 5). This procedure is efficient but it often fails to find any equivalent term.

### B. Technical Preliminaries

The goal in Program Synthesis is to generate functions that meet a set of specified constraints. These constraints can be formulated in terms of many-sorted second-order logic, using a set of non-empty sorts $S = \{\sigma_1, \sigma_2, ...\}$. If the function we aim to synthesize has a rank of $\sigma_1 \cdots \sigma_n\sigma$, then we can express the synthesis problem as:

$$\exists f : \sigma_1 \cdots \sigma_n\sigma, \forall \vec{x} : \sigma_1 \cdots \sigma_n. P[f, \vec{x}]$$

where $f$ is a second-order variable representing the target function, $\vec{x}$ is a tuple of first-order variables representing $f$'s input, and $P$ is a predicate that encodes the semantic constraints imposed on $f$ within a particular background theory $T$. In the context of SyGuS, additional syntactic constraints are imposed, specifically that the body of the synthesized function $f$ be in the language of a specified grammar $G$. We write $t \in \mathcal{L}(G)$ to denote that a term $t$ is in the language $\mathcal{L}(G)$ generated by a grammar $G$. In that case, we will also say that *$t$ is generated by $G$*.

In this paper, we consider a specific class of SyGuS problems in which a term $t[\vec{x}]$ over the free variables $\vec{x}$ that satisfies the semantic part of the synthesis problem is also provided. We can view this problem as the subclass of SyGuS problems in which the semantic specification has the form:

$$\exists f : \sigma_1 \cdots \sigma_n\sigma, \forall \vec{x} : \sigma_1 \cdots \sigma_n. f(\vec{x}) \approx t[\vec{x}].$$

The goal of our procedure, given $t$ and $G$, is to find a term $t'$ that is generated by $G$ and is equivalent to $t$ in the background theory $T$, denoted as $t' \approx_T t$.

Our procedure can be used in combination with approaches that are limited only to solving semantic specifications. We mentioned previous work [10] that introduced an efficient synthesis approach for single-invocation problems. This class of problems can be expressed by the conjecture:

$$\exists f : \sigma_1 \cdots \sigma_n\sigma, \forall \vec{x} : \sigma_1 \cdots \sigma_n. P[f(\vec{x}), \vec{x}] \quad (1)$$

where $P[f(\vec{x}), \vec{x}]$ is a first-order formula over the free variables $f, \vec{x}$. This conjecture is equivalent to the first-order formula:

$$\forall \vec{x} : \sigma_1 \cdots \sigma_n, \exists y : \sigma. P[y, \vec{x}] \quad (2)$$

A witness term $t[\vec{x}]$ for the quantifier in formula (1) can be constructed efficiently in practice from formula (2) by quantifier elimination (QE) if the theory $T$ admits QE or, more generally, from a refutation in $T$ of the negation of (2) [14]. However, finding then an equivalent term $t'$ generated by a specified grammar $G$ remains a challenge.

One approach is to use enumeration techniques developed for solving general SyGuS problems, however, those techniques do not scale well for large terms. Another option [10], is to match the structure of the term $t$ against the rules in $G$ to break down the synthesis problem into smaller, more manageable sub-problems. Unfortunately, this approach has its own limitations, since terms that slightly deviate from the grammar can be generated only by enumeration.

### C. Motivating Example

The following SyGuS problem demonstrates some of the shortcomings discussed above. We explain how to address them efficiently using matching and rewriting.

*Example* 1. Consider the synthesis problem

$$\forall x, y, u : \mathsf{Int}.\, f(x, y, u) \approx x - \mathsf{ite}(y + u > 0, y + u, 0)$$

over the theory of integers and let $t$ be the right-hand side of the equation. Assume our goal is to find an implementation of $f$ whose body is generated by a grammar $G$ with start symbol $A$ and the following production rules:

$$\begin{array}{lcl} A & \rightarrow & 0 \mid 1 \mid x \mid y \mid u \mid 0 - A \mid A + A \mid \mathsf{ite}(B, A, A) \\ B & \rightarrow & A \approx A \mid A > A \end{array}$$

The minimal solution to this problem, as measured by the length of its shortest derivation in the grammar, is prohibitively large for enumerative approaches, which typically explore the solution space by increasing term size. We observed that such approaches check 5K terms or more before finding a solution in this example, due to the combinatorial explosion in the set of terms generated by $G$ as a function of derivation length.

Alternatively, we can directly try to match $t$ with the right-hand side of production rules of the grammar, treating

non-terminals like $A$ and $B$ above as match variables. This makes it easy to detect, for instance, if $t$ is already in $\mathcal{L}(G)$. Unfortunately, this approach fails immediately in our example since $x - \mathsf{ite}(y+u > 0, y+u, 0)$ does not match any right-hand sides in $G$'s production rules.[1]

We can expand the possible patterns for term matching using rewriting modulo the background theory. Note, for example, that if we could add the rule $A \to A - A$ to the original grammar $G$, then matching $t$ against $A - A$ would lead to a solution, as both $x$ and $\mathsf{ite}(y + u > 0, y + u, 0)$ in turn match a rule for $A$. Now, we cannot add this rule directly as it would change the language of the grammar.[2] However, we can simulate doing so by considering the production rule $A \to A + (0 - A)$, derived from $A \to A + A$ and $A \to 0 - A$. This is because every term of the form $t_1 + (0 - t_2)$ is equivalent to the term $t_1 - t_2$ in the theory, something that can be easily shown using simple theory-specific rewrite rules. Based on this reasoning, we thus conclude that $x + (0 - \mathsf{ite}(y + u > 0, y + u, 0))$ is a solution for this example, being both equivalent to $t$ and generated by the provided grammar.

Note how this process is driven by matching against the grammar rules. This is much more direct than having to wait for terms to be constructed with enumerative methods. In the above example, the matching is made more flexible by utilizing term rewriting, which effectively provides a controlled form of matching modulo the background theory, loosening the limitations of relying just on the rules in the input grammar. Our experimental evaluation shows that this flexibility boosts the effectiveness of the matching-based approach considerably, as we discuss in Section III.

For convenience, we will call an expression derivable in a grammar $G$ (like the one in Example 1) a *pre-term (generated by G)* if it contains non-terminals. For instance, in Example 1, $0 - (A + A)$ is a pre-term, whereas $0 - (x + y)$ is not.

## II. A Procedure for SyGuS Solution Fitting

In this section, we describe and discuss our procedure for solution fitting in SyGuS, starting with a high-level overview. We then sketch the invariants that are maintained by the procedure, and briefly discuss its properties. In particular, the procedure is *solution sound*: it returns only terms that are indeed a solution of the given synthesis problem. The procedure is also *relatively terminating*: it is guaranteed to terminate with a (correct) solution if its underlying enumerative approach terminates with a solution.

### A. The Term Reconstruction Procedure

The reconstruction procedure, rcons, is described in Algorithm 1. It uses two main auxiliary procedures, match (Algorithm 2) and markSolved (Algorithm 3), to abstract the high-level structure from the finer details. We start by outlining the overall procedure and then elaborate on its key data structures and auxiliary functions. Finally, we illustrate through an example how the various components are integrated in rcons.

**Algorithm 1** $\mathsf{rcons}(G, t_0)$

**Require:** $t_0 : \mathsf{typeOf}(N_0)$ and $N_0$ is start symbol of $G$

```
1:  k_0 ← newVar(K, N_0)
2:  Obs ← {(k_0, t_0↓)}
3:  Pool, CandSols, Sol ← ∅, ∅, ∅
4:  Targets ← {(N_0, t_0↓)}
5:  while k_0 ∉ dom(Sol) do
6:      for non-terminal N of G do                ▷ Enumeration Phase
7:          s ← nextEnum(N)
8:          if FV(s) = ∅ then
9:              if there is (k, t) ∈ Obs s.t. t ≈_T s then
10:                 markSolved(k, s)
11:             else
12:                 k ← newVar(K, N)
13:                 Obs ← Obs ∪ {(k, s)}
14:                 markSolved(k, s)
15:             end if
16:         else
17:             Pool ← Pool ∪ {(N, s)}
18:             Targets' ← {match(t, s) | (N, t) ∈ Targets}
19:         end if
20:     end for
21:     while Targets' ≠ ∅ do                     ▷ Match Phase
22:         Targets ← Targets ∪ Targets'
23:         Targets' ←
24:         {match(t, s) | (N, t) ∈ Targets', (N, s) ∈ Pool}
25:     end while
26: end while
27: return Sol(k_0)
```

We assume two countably infinite sets $\mathcal{K}$ and $\mathcal{Z}$ of (meta-level) variables. The variables in $\mathcal{K}$ are placeholders for terms that require reconstruction within the grammar. Those in $\mathcal{Z}$ are matching variables; they represent holes in the patterns used for matching. Variables in $\mathcal{K}$ and $\mathcal{Z}$ are associated with two mappings, nonTerminalOf and typeOf, which respectively return a non-terminal in the given grammar $G$ and a type for the variable. We denote by $\mathsf{FV}(t)$ the set of variables occurring in term $t$. Note that for the purposes of the procedure, only elements of $\mathcal{K}$ and $\mathcal{Z}$ are considered variables. We also assume a *rewriting* procedure that rewrites each term $t$ to a term $t{\downarrow}$, the *rewritten form* of $t$. We require that the procedure be *sound*, that is, $t{\downarrow} \approx_T t$ for the background theory $T$, but not that it be confluent. This means that $t{\downarrow}$ and $s{\downarrow}$ may be distinct for some $T$-equivalent terms $t$ and $s$.

The procedure rcons updates the following *global* program variables.

- $Obs$ stores a set of pairs of the form $(k, t)$, where $k$ is a variable from $\mathcal{K}$ and $t$ is a term. We refer to variable $k$ as an *obligation* and say it is *closed* if we succeed in finding a term $t'$ equivalent to $t$ and generated by $\mathsf{nonTerminalOf}(k)$.
- $Pool$ stores a set of pairs of the form $(N, t)$, where $N$

[1] Note that it does not match with $0 - A$ because the terminal symbols $x$ and $0$ do not match.

[2] Observe that $A - A$ is not derivable from $A$ in the original grammar.

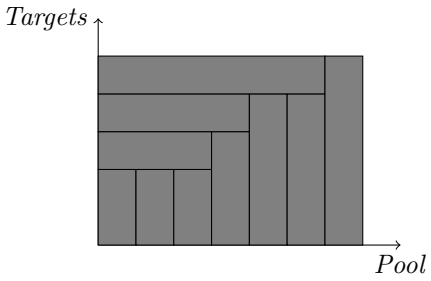


Fig. 1. The relationship between the set of patterns, $Pool$, and the set of terms to be reconstructed, $Targets$, can be depicted by the vertical and horizontal bars, respectively. The vertical bars represent the enumeration phases, while the horizontal bars represent the matching phases.

is a non-terminals of the grammar $G$ and $t$ is a term possibly with free variables from $\mathcal{Z}$. Those free variables in $t$ are used as holes for matching. If a match from an enumerated term against $t$ succeeds, then this match is used to generate further obligations.

- $CandSols$ stores a set of pairs of the form $(k, t)$, where $t$ is a potential solution to the obligation $k$. Terms $t$ in this mapping may contain variables from $\mathcal{K}$, meaning they may be built from terms corresponding to obligations that have yet to be closed.
- $Sol$ stores a mapping consisting of pairs $k \mapsto t$, where $t$ is an actual solution to the obligation $k$ for being a term generated by $\mathsf{nonTerminalOf}(k)$. We write $Sol(s)$ for the result of applying $Sol$ as a substitution to the term $s$.

The procedure rcons maintains a number of invariants over these sets, described in detail in Section II-D.

**Procedure** rcons**:** The main procedure, rcons, takes as input a term $t_0$ to be reconstructed in a grammar $G$. It first creates the main obligation to reconstruct, $(k_0, t_0{\downarrow})$, where $k_0$ is a variable used to refer to $t_0{\downarrow}$ in the procedure. The set of terms to reconstruct, $Targets$, is initialized with $t_0{\downarrow}$. The procedure is divided into two phases: the *enumeration phase* and the *match phase*.

In the *enumeration phase*, new patterns for each non-terminal in the grammar are enumerated using the iterator nextEnum. If a pattern $s$ returned by nextEnum is a ground term (i.e., $\mathsf{FV}(s) = \emptyset$), then we check if $s$ closes any obligation $k$. If it does, we call $\mathsf{markSolved}(k, s)$ to notify other candidate solutions that depend on $k$. Otherwise, a new obligation $(k, s)$ is created on the fly, and $s$, for being in $\mathcal{L}(G)$, becomes the solution to this obligation. This is done in case an equivalent term is encountered in the future. If the pattern $s$ is not a ground term, then it is matched against all the terms in $Targets$, and any potential subterms to reconstruct are saved in a new set $Targets'$. Additionally, $s$ is added to the set $Pool$ of patterns as it may be useful in matching against new terms to reconstruct in the match phase.

In the *match phase*, the new subterms in $Targets'$ are appended to the set $Targets$ and matched against the set of patterns that have been enumerated so far. Any subterms returned by this step are stored in $Targets'$ and the process is repeated until no new subterms are returned, indicating that all patterns have been matched against all terms.

**Algorithm 2** $\mathsf{match}(t, s)$

**Require:** $\mathsf{FV}(t) = \emptyset$, $\mathsf{FV}(s) \subseteq \mathcal{Z}$, $\mathsf{typeOf}(t) = \mathsf{typeOf}(s)$

```
1:  Targets' ← ∅
2:  if patternMatch(s↓, t) = σ then
3:      τ ← ∅
4:      for (z, st) ∈ σ do
5:          if there is (sk, t') ∈ Obs s.t. t' ≈_T st then
6:              Obs ← Obs ∪ {(sk, st)}
7:              τ ← τ ∪ {z ↦ sk}
8:          else
9:              SN ← nonTerminalOf(z)
10:             sk ← newVar(K, SN)
11:             τ ← τ ∪ {z ↦ sk}
12:             Obs ← Obs ∪ {(sk, st)}
13:             Targets' ← Targets' ∪ {(SN, st)}
14:         end if
15:         let k be s.t. (k, t) in Obs
16:         CandSols ← CandSols ∪ {k ↦ τ(s)}
17:         if FV(Sol(τ(s))) = ∅ then
18:             markSolved(k, τ(s))
19:         end if
20:     end for
21: end if
22: return Targets'
```

Once the matching phase is complete, the solution status of the main obligation $k_0$ is evaluated. If it is solved, the reconstruction process is complete. If not, then the current pool of enumerated patterns cannot derive a solution, and the enumeration phase is resumed, to generate new patterns.

The alternating behavior of the enumeration and matching phases is depicted in Figure 1. The vertical bars represent the enumeration phases, during which new patterns are added to $Pool$. It is possible for multiple enumeration phases to occur before a match is successful. Once a match succeeds, terms to be reconstructed are added to $Targets$, and the matching phase begins. The horizontal bars represent the matching phases, during which each new term is matched against all the patterns in $Pool$.

**Procedure** newVar**:** This subprocedure can be invoked with either $\mathcal{K}$ or $\mathcal{Z}$ to obtain a fresh variable $v$ for nonterminal $N$ from $\mathcal{K}$ or $\mathcal{Z}$, respectively. The procedure updates the mapping nonTerminalOf so that $\mathsf{nonTerminalOf}(v) = N$.

**Procedure** nextEnum**:** This is a stateful procedure, called on line 7 of the main procedure rcons, parametrized by a non-terminal $N$ of the grammar. Let $\mathcal{T}(N)$ be the set of all terms and pre-terms generated by $N$. Each call to $\mathsf{nextEnum}(N)$ returns the next term in a (fair) enumeration of the set obtained from $\mathcal{T}(N)$ by replacing all occurrences of non-terminals with variables from $\mathcal{Z}$. For non-terminal $A$ in the grammar of Example 1, this set of terms would include, for instance, $0-z_1$, $z_1 + z_2$, and $\mathsf{ite}(z_1, z_2, z_3)$ where $z_1, z_2, z_3 \in \mathcal{Z}$. In practice, nextEnum is implemented by modifying standard methods for

**Algorithm 3** markSolved$(k, s)$

**Require:** $\mathsf{FV}(s) = \emptyset$ and $s \approx_T t$ for some $(k, t) \in Obs$.

1: $Sol \leftarrow Sol \cup \{(k, s)\}$
2: $CandSols \leftarrow CandSols \cup \{(k, s)\}$
3: **repeat**
4: $\quad CandSols' \leftarrow CandSols$
5: $\quad CandSols \leftarrow \{(k, Sol(s)) \mid (k, s) \in CandSols'\}$
6: $\quad$ **for** $(k, s) \in CandSols$ **do**
7: $\quad\quad$ **if** $k \notin dom(Sol)$ **then**
8: $\quad\quad\quad Sol \leftarrow Sol \cup \{k \mapsto s\}$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: **until** $CandSols' = CandSols$

fast term enumeration [9].

The terms returned by nextEnum serve two purposes in the main procedure: those with no free variables from $\mathcal{Z}$ are used to discharge terms to synthesize, on lines 9-15 (as we will see later in subprocedure markSolved); those with free variables from $\mathcal{Z}$ are used as new patterns for matching on lines 17-18.

**Procedure** match **(Algorithm 2):** This function takes in a term $t$ to be reconstructed and a pattern $s$. The term $t$ is assumed to be in rewritten form (i.e., $t = t{\downarrow}$), However, this is not necessarily the case for $s$ as it must preserve the syntactic structure enforced by the grammar.

The first step is to rewrite $s$ and attempt to match the structure of $t$. If the match succeeds, match returns the substitution $\sigma$ (from variables in $\mathcal{Z}$ to subterms from $t$) required to unify $s{\downarrow}$ and $t$. This substitution represents the subterms that must be synthesized before $s$ can be marked as a solution. For each pair $(z, st)$ in $\sigma$, match finds a corresponding $sk$ in $Obs$ or creates a new obligation $sk$ and adds the pair $(sk, st)$ to $Obs$. match then creates a new substitution $\tau$ from $z$ to $sk$ and applies it to $s$ to construct a candidate solution, $\tau(s)$, for $k$ (the variable whose obligation is $t$). If there are no subterms to synthesize, $\tau(s)$ is a solution to $k$, and markSolved$(k, \tau(s))$ is called.

*Example* 2. Consider the following obligations and candidate solution sets: $Obs = \{(k_0, x - y), (k_1, x)\}$ and $CandSols = \{(k_1, x)\}$. If we invoke match with $s = z_0 + (0 - z_1)$ and $t = x - y$, then matching will succeed (assuming $s{\downarrow} = z_0 - z_1$) and return a substitution $\sigma = \{z_0 \mapsto x, z_1 \mapsto y\}$. The check at line 5 will determine that $x$ is already in $Obs$, so there is no need to create a new obligation for it. However, $y$ is not in $Obs$, so a new obligation $(k_2, y)$ will be added.

As we process the substitutions in $\sigma$, we construct a new substitution $\tau = \{z_0 \mapsto k_1, z_1 \mapsto k_2\}$ and apply it to $s$ to construct a candidate solution for $k_0$, namely $k_1 + (0 - k_2)$. The match procedure returns $\{(A, y)\}$ as the set of subterms to reconstruct, resulting in the following updated sets:

$$Obs = \{(k_0, x - y), (k_1, x), (k_2, y)\}$$
$$CandSols = \{(k_1, x), (k_0, k_1 + (0 - k_2))\}$$

**Procedure** markSolved **(Algorithm 3):** The set $CandSols$ contains candidate solutions $s$ to obligations $k$. Specifically, term $s$ is a solution if it does not contain variables. Whenever that is the case, markSolvedis called to update all other potential solutions that depend on $k$. During this process, complete solutions for other obligations may be discovered. Thus, we repeat this step until no further such terms appear (i.e., a fixed point is reached). Algorithm 3 provides a basic implementation of this procedure.

*Example* 3. As an illustration, consider the following scenario, which uses the grammar from Example 1. Let $CandSols = \{(k_0, x + k_1), (k_1, 0 - k_2)\}$ and $Sol = \emptyset$. Invoking markSolved$(k_2, y)$ results in:

$$\begin{aligned} CandSols = \{&(k_0, x + k_1), (k_1, 0 - k_2), \\ &(k_2, y), (k_1, 0 - y), (k_0, x + (0 - y))\} \\ Sol = \{&k_0 \mapsto x + (0 - y), k_1 \mapsto 0 - y, k_2 \mapsto y\} \end{aligned}$$

### B. Rewrite Rule Discovery

There are two situations in which our overall procedure utilizes theory reasoning to synthesize new rewrite rules. The first instance occurs in line 9 of Algorithm 1, where we aim to establish if a ground term we derived from the grammar is $T$-equivalent to one of the target terms in $Obs$ that we want to synthesize. The second instance occurs in line 5 of Algorithm 2 after a match against a pattern is successful and generates a substitution. We again utilize theory reasoning there to determine if there are any obligations that are $T$-equivalent to the substitution that match asks us to synthesize. This line creates an equivalence class of terms to synthesize, where synthesizing any term in the class amounts to synthesizing all of them.

We use an SMT solver to discover new rewrite rules. Calling the solver every time to find a term's equivalence class, however, is inefficient. Instead, we follow the approach in Nötzli et al. [15] and build a trie whose leaves are equivalence classes and whose nodes are points where the equivalence classes differ. Those points can be obtained by requesting a model when the SMT solver determines that two terms are not $T$-equivalent.

Note that calling the SMT solver is more costly than just calling the rewriter, but doing so leads to larger equivalence classes, thereby providing a greater selection of terms for reconstruction. In particular, simpler terms may become available, which can greatly accelerate the reconstruction process. Although the approach in Nötzli et al. [15] reduces the number of solver calls required, some calls may still take too long. In practice then, we set a time limit for each call and conservatively assume non-equivalence when a call times out.

### C. Revisiting the Motivating Example

We return now to the motivating example from Section I-C to give a detailed run-through of the rcons procedure, which we simplify slightly to keep the presentation manageable. The

objective is to reconstruct the term $t_0 = x - \mathsf{ite}(y > 0, y, 0)$ to an equivalent one in the language of the grammar below.

$$A \to 0 \mid 1 \mid x \mid y \mid 0 - A \mid A + A \mid \mathsf{ite}(B, A, A)$$
$$B \to A \approx A \mid A > A$$

rcons starts by initializing the data structure values with:

$$Obs = \{(k_0, t_0)\} \qquad Pool = CandSols = Sol = \emptyset$$

where $k_0$ is a fresh variable. During the enumeration phase, rcons begins by enumerating the set of terminal symbols of the grammar: 0, 1, $x$, and $y$. None of them match with $t_0$, but they are assigned as solutions to artificial obligations, in case they turn out to be equivalent to subterms of $t_0$.

$$Obs = \{(k_0, t_0), (k_1, 0), (k_2, 1), (k_3, x), (k_4, y)\}$$
$$CandSols = \{(k_1, 0), (k_2, 1), (k_3, x), (k_4, y)\}$$
$$Sol = \{k_1 \mapsto 0, k_2 \mapsto 1, k_3 \mapsto x, k_4 \mapsto y\}$$

The procedure then proceeds to the next stage of enumeration, deriving more complex patterns from $\mathcal{T}(A)$ and $\mathcal{T}(B)$. The patterns that can be instantiated are stored in $Pool$. For brevity, we explicitly list only the significant patterns below.

$$Pool = \{(A, 0 - z_0), (A, z_1 + z_2), (A, \mathsf{ite}(z_3, z_4, z_5)), (B, z_6 > z_7), ...\}$$

rcons continues its process of deriving patterns from the grammar in an effort to reconstruct $t_0$. At some point, it arrives at the pattern $z_8 + (0 - z_9)$, whose rewritten form is $z_8 - z_9$, matching $t_0$. The match creates a substitution $\sigma = \{z_8 \mapsto x, z_9 \mapsto \mathsf{ite}(y > 0, y, 0)\}$, mapping variables from $\mathcal{Z}$ to subterms to reconstruct. Since we do not know how to reconstruct all subterms yet, match creates another substitution, $\tau = \{z_8 \mapsto k_3, z_9 \mapsto k_5\}$, which replaces the subterms with (potentially new) corresponding obligations. Substitution $\tau$ is then applied to $s$ to create a candidate solution $\tau(z_8 + (0 - z_9)) = k_3 + (0 - k_5)$ for $k_0$. Now we have:

$$Obs = \{(k_0, t_0), (k_1, 0), (k_2, 1), (k_3, x), (k_4, y), ..., (k_5, \mathsf{ite}(y > 0, y, 0))\}$$
$$Pool = \{..., (A, \mathsf{ite}(z_3, z_4, z_5)), (B, z_6 > z_7), ..., (A, z_8 + (0 - z_9))\}$$
$$CandSols = \{(k_1, 0), (k_2, 1), (k_3, x), (k_4, y), ..., (k_0, k_3 + (0 - k_5))\}$$
$$Sol = \{k_1 \mapsto 0, k_2 \mapsto 1, k_3 \mapsto x, k_4 \mapsto y, ...\}$$

At this point, match returns $\{(A, \mathsf{ite}(y > 0, y, 0))\}$, the new term to synthesize. Since $Targets'$ is not empty, rcons enters the match phase for the first time, matching $\mathsf{ite}(y > 0, y, 0)$ against all patterns stored in $Pool$. Matching against $\mathsf{ite}(z_3, z_4, z_5)$ succeeds and generates the substitutions below.

$$\sigma = \{z_3 \mapsto y > 0, z_1 \mapsto y, z_2 \mapsto 0\}$$
$$\tau = \{z_3 \mapsto k_6, z_4 \mapsto k_4, z_5 \mapsto k_1\}$$

match then adds $\tau(\mathsf{ite}(z_3, z_4, z_5)) = \mathsf{ite}(k_6, k_4, k_1)$ as a candidate solution for $k_5$.

This process is then repeated for the term $y > 0$. This time, rcons matches $y > 0$ against $z_6 > z_7$ generating the candidate solution $k_4 > k_1$. Since both $k_1$ and $k_4$ are solved obligations, $\mathsf{markSolved}(k_6, k_4 > k_1)$ is invoked to construct complete solutions for $k_6$ and any other obligations that depend on it, such as $k_5$ and $k_0$.

$$Obs = \{(k_0, t_0), (k_1, 0), (k_2, 1), (k_3, x), (k_4, y), ..., (k_5, \mathsf{ite}(y > 0, y, 0)), (k_6, y > 0)\}$$
$$CandSols = \{(k_1, 0), (k_2, 1), (k_3, x), (k_4, y), ..., (k_0, k_3 + (0 - k_5)), (k_5, \mathsf{ite}(k_6, k_4, k_1)), (k_6, k_4 > k_1), ...\}$$
$$Sol = \{k_0 \mapsto x + (0 - \mathsf{ite}(y > 0, y, 0)), ...\}$$

With the solution to $k_0$ contained in $Sol$, rcons exits the main loop and returns $Sol(k_0) = x + (0 - \mathsf{ite}(y > 0, y, 0))$.

### D. Properties

Procedure rcons maintains several invariants that are essential to its correctness. Those invariants are listed below. We say that a substitution $\sigma$ *respects grammar* $G$ if for all $v \mapsto t \in \sigma$, term $t$ is generated by $\mathsf{nonTerminalOf}(v)$ of $G$.

**Invariant 1.** $Obs$ is a set of pairs of the form $(k, t)$, where $k$ is an obligation to reconstruct the term $t$ in the grammar, such that:

1) $t : \mathsf{typeOf}(k)$;
2) $\mathsf{FV}(t) = \emptyset$;
3) $s \approx_T t$ for all $(k, s) \in Obs$.

**Invariant 2.** $Pool$ is a set of pairs of the form $(N, t)$, where $t$ is a pattern shared by some terms generated from the non-terminal $N$, such that:

1) $t : \mathsf{typeOf}(N)$;
2) $\mathsf{FV}(t) \subseteq \mathcal{Z}$;
3) the ground term $\sigma(t)$ is generated by $N$ for all substitutions $\sigma$ over $\mathcal{Z}$ that respect grammar $G$.

**Invariant 3.** $CandSols$ is a set of pairs of the form $(k, t)$, where $t$ is a potential solution to the obligation $k$, such that:

1) $t : \mathsf{typeOf}(k)$;
2) $\mathsf{FV}(t) \subseteq \mathcal{K}$;
3) The ground term $\tau(t)$ is generated by $\mathsf{nonTerminalOf}(k)$ for all substitutions $\tau$ over $\mathcal{K}$ that respect grammar $G$;
4) if $t$ is ground, $s \approx_T t$ for all $(k, s) \in Obs$.

**Invariant 4.** $Sol$ is a mapping with pairs of the form $k \mapsto t$, where $t$ is a solution to the obligation $k$, such that:

1) $t : \mathsf{typeOf}(k)$;
2) $\mathsf{FV}(t) = \emptyset$;
3) $t$ is generated by $\mathsf{nonTerminalOf}(k)$;
4) $s \approx_T t$ for all $(k, s) \in Obs$.

*Correctness:* The main correctness property of procedure rcons can be summarized as follows.

**Proposition 1.** *If* $\mathsf{rcons}(G, t_0)$ *successfully terminates with a solution term* $t$*, then:*

1) $t \approx_T t_0$, *and*
2) $t$ *is generated by the start symbol* $N_0$ *of grammar* $G$.

Thus, the returned term $t$ is semantically equivalent to the input term $t_0$ and satisfies the syntactic restrictions $G$ as well.

*Proof sketch.* One can show that each update in rcons of the globals sets maintains invariants 1–4. Points (1) and (2) above follow from those invariants. In particular, the returned solution $t$ is $Sol(k_0)$, where $(k_0, t_0{\downarrow}) \in Obs$ due to the initial value of $Obs$. By Invariant 4, $t \approx_T t_0{\downarrow} \approx_T t_0$. Furthermore, by Invariant 3, we have that $Sol(k_0)$ is generated by $\mathsf{nonTerminalOf}(k_0) = N_0$. □

*Termination:* We briefly remark on the termination of rcons by examining the enumeration and match phases. We argue that the procedure terminates whenever a term $t \approx_T t_0$ can be produced by enumeration from the grammar, and the underlying theory $T$ has a decision procedure for term equivalence. Even if the matching components of the procedure fail to make progress, the enumerator will still arrive at a solution since enumeration approaches terminate in this setting. So, it is sufficient to show that both phases of the procedure terminate. The enumeration phase terminates because $T$-equivalence checks terminate. In the match phase, every call to match returns smaller subterms as it only matches against the syntactic structure of the terms to synthesize. As a result, eventually, the algorithm will reach a point where matching fails or the terms can no longer be broken down further. In either case, match returns an empty set and the match phase terminates.

*Unrealizability:* It is possible that $t_0$ may be *unrealizable* with respect to $G$, meaning that there is no term $t \in \mathcal{L}(G)$ such that $t \approx_T t_0$. In its current version, rcons lacks a mechanism to detect and report unrealizability; it simply diverges. However, we can draw upon techniques from existing enumeration approaches, which rcons builds upon, to detect simple cases. An alternative approach would involve checking the realizability of syntactic constraints during the matching phase, potentially requiring a recursive invocation of SyGuS on sub-target terms. However, for performance reasons and due to rcons not being designed for handling this situation, we do not incorporate these mechanisms. For a more comprehensive treatment of unrealizability, we refer readers to the work by Hu et al. [16].

## III. Experimental Evaluation

We have implemented the rcons procedure as a module within CVC5's synthesis solver. Our implementation is comprised of roughly 800 lines of C++ code. We evaluated it on two classes of SyGuS benchmarks: the first was designed to assess the procedure's performance under a range of grammars and theories, while the second consists of benchmarks from the SyGuS 2019 competition for which the procedure directly applies. The benchmarks for both classes had a common structure consisting of four components:

1) Miscellaneous declarations and definitions.
2) The function $f$ to synthesize.
3) Syntactic constraints on $f$ via a grammar $G$.
4) One semantic constraint of the form $\forall \vec{x}.\, f(\vec{x}) \approx t[\vec{x}]$, where $t$ is the solution we want to fit into $G$.

In evaluating the effectiveness of rcons, we compared its performance against the winning solvers of previous SyGuS competitions: **EUSolver**[3] [6] and CVC4 [17] (and its latest iteration, CVC5 [18]). We used **EUSolver** in its default configuration. We considered purely enumerative configurations of CVC4 and CVC5 (referred to as **cvc4-enum** and **cvc5-enum**, respectively), the procedure from Section 5 of [10] (**cvc4-rcons**), as well as our new rcons procedure (**cvc5-rcons**). An additional configuration, **cvc5-match**, uses a modified version of rcons, with the enumeration phase disabled, and $Pool$ only holding the patterns found in the given grammar. This configuration builds the solution solely through matching, serving as a baseline for our comparisons.

We ran our comparative evaluation on the StarExec platform [19], with a time limit of 30 minutes per instance.

### A. Crafted Benchmarks

In the first set of experiments, we randomly generated SyGuS solution refinement benchmarks for three SMT-LIB theories of interest: bit-vectors, integer arithmetic, and strings. We followed the steps below to craft the benchmarks:

1) We constructed a reference grammar for each of the three theories, each comprising a majority of the symbols for that specific theory.[4] We used the reference grammars to craft our set of benchmarks.
2) We developed a procedure to generate random terms from the reference grammars, with the derivation length of the terms adhering to a geometric distribution.
3) We used the reference grammars to construct random grammars containing the original non-terminals (with modified rules) and new non-terminals.

Using the simple grammar $A \to 1 \mid x \mid y \mid A + A$ as an example, the procedure used in Step 2 above works as follows. A string containing only the start symbol, $A$, is first generated. A coin is then flipped to determine whether or not to replace a non-terminal with a randomly selected rule containing non-terminals (only $A+A$ in our case). For example, if the first two coin flips yield heads, then $A$ will be replaced with $A+A$ and either the first or second $A$ (randomly chosen) with $A + A$. This would result in either $(A + A) + A$ or $A + (A + A)$. The process continues until a tails is seen, at which point all remaining non-terminals are replaced with randomly selected terminal symbols ($0$, $x$, or $y$ in our case) and the resulting SyGuS term (e.g., $0 + (x + x)$) is returned. To ensure that the derivation length of the terms follows a geometric distribution, we crafted the reference grammars so that each non-terminal symbol has at least one terminal rule and at least one rule containing a non-terminal symbol.

[3]We updated **EUSolver** with bug fixes and added missing support for theory symbols to conform to the latest revisions of SMT-LIB's standard and theories.

[4]Some redundant theory symbols were omitted for simplicity. Our reference grammars along with the benchmarks we considered are available at https://github.com/cvc5/artifact-fmcad23-sygus.

| Solver\Fragment | bv (1K) | nia (1K) | slia (1K) | Total (3K) |
|---|---|---|---|---|
| cvc4-enum | 518 | 90 | 248 | 856 |
| cvc4-rcons | 427 | 113 | 66 | 606 |
| cvc5-enum | 530 | 222 | 305 | 1057 |
| cvc5-match | 17 | 6 | 6 | 29 |
| cvc5-rcons | 955 | 810 | 604 | 2369 |
| EUSolver | 498 | 243 | 405 | 1146 |

TABLE I
THE TABLE SHOWS THE NUMBER OF SYNTHESIZED BENCHMARKS SOLVED BY EACH SOLVER CONFIGURATION.

The construction in Step 3 works as follows. First, rules containing non-terminals ($A + A$ for $A$ in the example) are examined. A coin is flipped to decide whether or not to add this rule to the random rules for $A$ in the new grammar. If the rule is added, each non-terminal within the rule is examined and either kept or replaced with a different (potentially new) non-terminal of the same type. We add at least one terminal rule for each non-terminal to ensure that the random grammar is well defined. Those two steps are repeated for each original and new non-terminal. The procedure is forced to terminate by making the probability of adding new non-terminals inversely proportional to the number of existing non-terminals. An example of a random grammar that can be generated from the grammar above is:

$$A \rightarrow x \mid A_1 + A_2 \qquad A_1 \rightarrow 1 \mid A_2 + A_3$$
$$A_2 \rightarrow x \mid y \mid A_1 + A \qquad A_3 \rightarrow x \mid A_1 + A_3$$

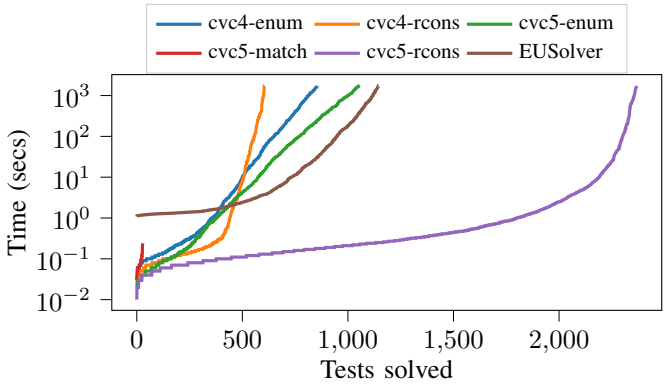


Fig. 2. Cactus plot comparing performance of solvers on crafted benchmarks.

*Results:* The results presented in Table I demonstrate a marked superiority of our new implementation compared to other solver configurations. Specifically, **cvc5-rcons** was able to successfully solve 846 benchmarks that were not solved by any other configuration. Conversely, only 44 benchmarks were solved by other configurations but not by **cvc5-rcons**. This disparity becomes even more pronounced when comparing **cvc5-rcons** against individual configurations, such as 1,334 vs. 22 uniquely solved benchmarks for **cvc5-enum** and 1,256 vs. 33 for **EUSolver**.

Furthermore, our rcons procedure consistently exhibits faster performance when solving commonly solved benchmarks. This is particularly pronounced in the case of bit-vector benchmarks, where significant speedups of one order of magnitude are observed (16, 17, and 74 times faster than **EUSolver**, **cvc4-rcons**, and **cvc5-enum** respectively). While more modest, still significant speedups are observed in the theories of integers (up to 21 times) and strings (up to 13 times), this is largely due to the other solvers timing out on most benchmarks in those fragments. The cactus plot presented in Figure 2 provides a summary of the results.

| Solver\Fragment | General (877) |
|---|---|
| cvc4-enum | 438 |
| cvc4-rcons | 205 |
| cvc5-enum | 672 |
| cvc5-match | 82 |
| cvc5-rcons | 721 |
| EUSolver | 640 |

TABLE II
THE TABLE SHOWS THE NUMBER OF SYGUS 2019 COMPETITION BENCHMARKS SOLVED BY EACH SOLVER CONFIGURATION.

The number of benchmarks solved by **cvc5-match** is negligible when compared to that of the other configurations. This result shows that relying solely on the patterns provided in the grammar is not effective and generating new patterns through enumeration is critical for the success of rcons.

### B. SyGuS Competition Benchmarks

We also evaluated rcons on a subset of the SyGuS benchmarks [20], a set of 877 benchmarks used in previous SyGuS competitions [21], [22], [12], [13], which come from a variety of user applications. The subset contains only SyGuS solution fitting problems, the focus of rcons.

The results are shown in Table II and Figure 3. Again, **cvc5-rcons** outperformed the competition. In particular, it managed to solve 49 and 81 more benchmarks than **cvc5-enum** and **EUSolver**, respectively. Overall, **cvc5-rcons** solved 42 benchmarks that were not solved by any other solver.

The cactus plot from Figure 3 provides further evidence of the robustness of our approach with respect to previous solutions. In particular, the graph shows that a previous approach for matching and enumeration (**cvc4-rcons**) is able to solve many benchmarks quickly, but is eventually eclipsed in performance by **cvc4-enum**. In contrast, **cvc5-rcons** solves an even larger percentage of benchmarks quickly and continues to compete with **cvc5-enum** for the entire 30 minute timeout. We note that **cvc5-enum** performs well in this set of benchmarks because 66% of it consists of circuit synthesis problems, which are not well-suited for unification and matching approaches. Nevertheless, **cvc5-rcons** still surpasses **cvc5-enum** by solving 25 additional problems in this category.

### C. Key Insights

Our analysis reveals two factors that have a significant impact on the performance of rcons across both benchmark categories.

The first significant factor concerns the extent of rewrites supported by a particular theory. The presence of an increased number of rewrite rules often allows rcons to generate solutions that markedly differ from those produced by enumerative

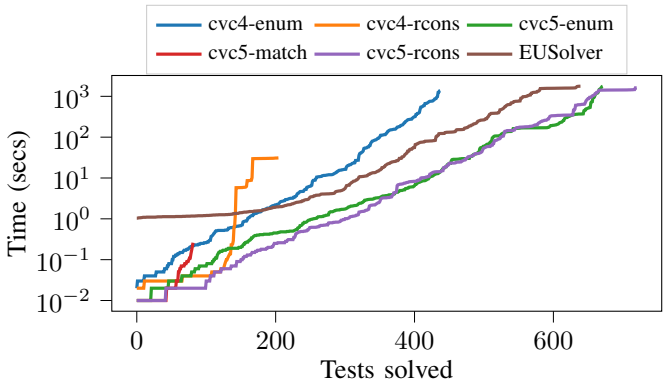


Fig. 3. Cactus plot comparing performance of solvers on SyGuS Competition benchmarks.

| **Parameter** | **General** (877) |
|---|---|
| 0.5 | 643 |
| 0.9 | 616 |
| 0.99 | 634 |
| 0.999 | 643 |
| 0.9999 | 721 |
| 0.99999 | 718 |

TABLE III
THE TABLE SHOWS THE NUMBER OF SYGUS 2019 COMPETITION BENCHMARKS SOLVED BY **CVC5-RCONS** WITH DIFFERENT PARAMETERS.

approaches. For instance, within Table I, we observe that **cvc5-rcons** outperforms alternative solver configurations in the bit-vector benchmarks. This enhanced performance is primarily attributed to the prevalence of rewrite rules within this theory. Conversely, when dealing with circuit synthesis benchmarks, where meaningful boolean rewrite rules are lacking, **cvc5-rcons** exhibits only marginal improvements over **cvc5-enum**, as it frequently converges to solutions identical to those found by **cvc5-enum**.

The second significant factor revolves around the number of patterns employed. The core principle guiding rcons is to mitigate the constraints imposed by the provided grammar rules by producing patterns that offer greater flexibility for matching purposes. Nonetheless, an excessive number of patterns can lead to extended durations in the match phase, thereby degrading overall performance. Our approach involves initially generating a substantial number of patterns and subsequently transitioning to enumerating ground terms. The optimal number of patterns depends on the specific grammar and theory. In our implementation, we leverage a geometric distribution to regulate pattern generation. Table III underscores the substantial impact of varying the geometric distribution parameter on the performance of **cvc5-rcons**.

## IV. CONCLUSION AND FUTURE WORK

We have presented a novel procedure for the SyGuS solution fitting problem. The procedure enabled the development of an advanced enumerative solver that significantly outperforms other state-of-the-art SyGuS solvers. Our experimental results show that our procedure finds solutions efficiently by deriving complex patterns through enumeration, and using them for matching. The procedure is not restricted to a particular background theory and can be used in combination with any theory solver that supports rewrites and equivalence checks.

We conjecture that the scalability of our procedure can be leveraged in synthesis problems involving optimization constraints. One class of problems is software optimization, as applied in compilers for embedded SQL queries, linear algebra operations, and circuit synthesis [23], [24], [25]. Current approaches based on synthesis rely on enumerative techniques to generate optimal programs, which does not scale well. A more practical approach could be to first synthesize an initial program, which may not be as efficient as the optimal one, and then gradually optimize it by optimizing its subterms. Although this does not always guarantee optimal performance, it is much more scalable for larger programs. We plan to investigate enhancements to the rcons procedure to handle weighted grammars and support this use case.

## REFERENCES

[1] R. Alur, M. M. K. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing finite-state protocols from scenarios and requirements," in *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings* (E. Yahav, ed.), vol. 8855 of *Lecture Notes in Computer Science*, pp. 75–91, Springer, 2014.

[2] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5-6, pp. 475–495, 2013.

[3] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006* (J. P. Shen and M. Martonosi, eds.), pp. 404–415, ACM, 2006.

[4] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5-6, pp. 497–518, 2013.

[5] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Dependable Software Systems Engineering* (M. Irlbeck, D. A. Peled, and A. Pretschner, eds.), vol. 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pp. 1–25, IOS Press, 2015.

[6] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I* (A. Legay and T. Margaria, eds.), vol. 10205 of *Lecture Notes in Computer Science*, pp. 319–336, 2017.

[7] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020* (A. F. Donaldson and E. Torlak, eds.), pp. 1159–1174, ACM, 2020.

[8] S. Padhi, R. Sharma, and T. D. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016* (C. Krintz and E. D. Berger, eds.), pp. 42–56, ACM, 2016.

[9] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, and C. Tinelli, "cvc4sy: Smart and fast term enumeration for syntax-guided synthesis," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II* (I. Dillig and S. Tasiran, eds.), vol. 11562 of *Lecture Notes in Computer Science*, pp. 74–83, Springer, 2019.

[10] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett, "Counterexample-guided quantifier instantiation for synthesis in SMT," in *Computer Aided Verification - 27th International Conference, CAV*

*2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II* (D. Kroening and C. S. Pasareanu, eds.), vol. 9207 of *Lecture Notes in Computer Science*, pp. 198–216, Springer, 2015.

[11] R. Alur, P. Cerný, and A. Radhakrishna, "Synthesis through unification," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II* (D. Kroening and C. S. Pasareanu, eds.), vol. 9207 of *Lecture Notes in Computer Science*, pp. 163–179, Springer, 2015.

[12] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "SyGuS-Comp 2016: Results and analysis," in *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016* (R. Piskac and R. Dimitrova, eds.), vol. 229 of *EPTCS*, pp. 178–202, 2016.

[13] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "SyGuS-Comp 2017: Results and analysis," in *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017* (D. Fisman and S. Jacobs, eds.), vol. 260 of *EPTCS*, pp. 97–115, 2017.

[14] A. Reynolds, V. Kuncak, C. Tinelli, C. W. Barrett, and M. Deters, "Refutation-based synthesis in SMT," *Formal Methods Syst. Des.*, vol. 55, no. 2, pp. 73–102, 2019.

[15] A. Nötzli, A. Reynolds, H. Barbosa, A. Niemetz, M. Preiner, C. W. Barrett, and C. Tinelli, "Syntax-guided rewrite rule enumeration for SMT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings* (M. Janota and I. Lynce, eds.), vol. 11628 of *Lecture Notes in Computer Science*, pp. 279–297, Springer, 2019.

[16] Q. Hu, J. Breck, J. Cyphert, L. D'Antoni, and T. W. Reps, "Proving unrealizability for syntax-guided synthesis," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (I. Dillig and S. Tasiran, eds.), vol. 11561 of *Lecture Notes in Computer Science*, pp. 335–352, Springer, 2019.

[17] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, 2011.

[18] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I* (D. Fisman and G. Rosu, eds.), vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442, Springer, 2022.

[19] A. Stump, G. Sutcliffe, and C. Tinelli, "Starexec: A cross-community infrastructure for logic solving," in *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings* (S. Demri, D. Kapur, and C. Weidenbach, eds.), vol. 8562 of *Lecture Notes in Computer Science*, pp. 367–373, Springer, 2014.

[20] S. Padhi, A. Reynolds, A. Udupa, and E. Polgreen, "SyGuS benchmarks." https://github.com/SyGuS-Org/benchmarks, 2019.

[21] R. Alur, D. Fisman, S. Padhi, R. Singh, and A. Udupa, "SyGuS-Comp 2018: Results and analysis," *CoRR*, vol. abs/1904.07146, 2019.

[22] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Results and analysis of SyGuS-Comp'15," in *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015* (P. Cerný, V. Kuncak, and P. Madhusudan, eds.), vol. 202 of *EPTCS*, pp. 3–26, 2015.

[23] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, vol. 2 of *The Kluwer International Series in Engineering and Computer Science*. Springer, 1984.

[24] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA* (A. O. Mendelzon and J. Paredaens, eds.), pp. 34–43, ACM Press, 1998.

[25] M. Willsey, Y. R. Wang, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, "egg: Easy, efficient, and extensible e-graphs," *CoRR*, vol. abs/2004.03082, 2020.