CVC4SY: Smart and Fast Term Enumeration for Syntax-Guided Synthesis

Andrew Reynolds[1], Haniel Barbosa[1], Andres Nötzli[2], Clark Barrett[2], and Cesare Tinelli[1]

[1] The University of Iowa, Iowa City, USA
[2] Stanford University, Stanford, USA

Abstract. We present CVC4SY, a syntax-guided synthesis (SyGuS) solver based on three bounded term enumeration strategies. The first encodes term enumeration as an extension of the quantifier-free theory of algebraic datatypes. The second is based on a highly optimized brute-force algorithm. The third combines elements of the others. Our implementation of the strategies within the satisfiability modulo theories (SMT) solver CVC4 and a heuristic to choose between them leads to significant improvements over state-of-the-art SyGuS solvers.

1 Introduction

Syntax-guided synthesis (SyGuS) [3] is a recent paradigm for program synthesis, successfully used for applications in formal verification and programming languages. Most SyGuS solvers perform counterexample-guided inductive synthesis (CEGIS) [16]: a refinement loop in which a learner proposes solutions, and a verifier, generally a satisfiability modulo theories (SMT) solver [8, 9], checks them and provides counterexamples for failures. Generally, the learner enumerates some set of terms, while pruning spurious ones [17]. The simplicity and efficacy of enumerative SyGuS have made it the de facto approach for SyGuS, although alternatives exist for restricted fragments [4, 14].

In previous work [14], we have shown how the SMT solver CVC4 [5] can itself act as an efficient synthesizer. This tool paper focuses on recent advances in the enumerative subsolver of CVC4, culminating in the current SyGuS solver CVC4SY. Figure 1 shows its main components. The term enumerator is parameterized by an enumeration strategy chosen before solving: CVC4SY_S, whose constraint-based (smart) enumeration allows for numerous optimizations (Section 2); CVC4SY_F, based on a new approach for (fast) enumerative synthesis (Section 3), which has significant advantages with respect to the enumerative solver CVC4SY_S and other state-of-the-art approaches; and CVC4SY_H, based on a hybrid approach combining smart and fast enumeration (Section 4). All strategies are fully integrated in CVC4, meaning they support inputs in many background theories, including arithmetic, bit-vectors, strings, and floating point. We evaluate these approaches on a large set of benchmarks (Section 5).

The Problem A syntax-guided synthesis problem for a function f in a background theory T consists of a set of semantic restrictions, or specification, for f given by a (second-order) T-formula of the form $\exists f.\, \varphi[f]$, and a set of syntactic restrictions on

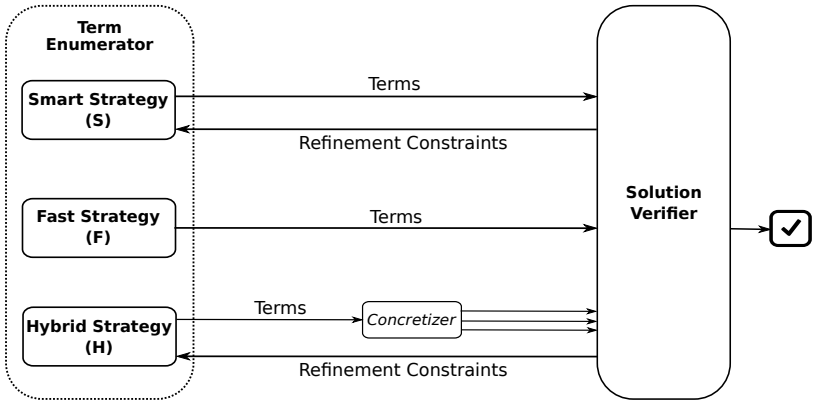

Fig. 1. Architecture of CVC4SY.

the solutions for f, typically expressed as a context-free grammar. An *enumerative* approach to this problem combines a *term enumerator* and a *solution verifier* for solving synthesis conjectures. The role of the term enumerator is to output a stream of terms $t_1, t_2, \ldots$ over some tuple $\bar{x}$ of variables representing the inputs of f, where each $t_i[\bar{x}]$ is a candidate solution. The role of the solution verifier is to check for each t_i whether it is a solution for f by determining if the negated conjecture $\neg\varphi[\lambda\bar{x}.t_i]$ is unsatisfiable.

Bounded term generation considers terms based on an ordering such as term size (the number of non-nullary symbols in a term). For each $k = 0, 1, 2, \ldots$, the term enumerator outputs a *finite* set S_k of terms, each of size at most k. Bounded term generation in CVC4SY is *complete* in the sense that, for any k, if f has a solution of size at most k, then at least one of the terms in S_k is a solution for f. The effectiveness of an approach for (complete) bounded term generation can be evaluated based on two criteria: (i) the number of terms it generates and (ii) the rate at which it generates them.

We follow two approaches for enumerative SyGuS in CVC4SY, each optimized for one of the criteria above: a *smart* approach and a *fast* one. The first aims to generate reasonably quickly the smallest set of terms while maintaining completeness, while the second aims to generate terms as quickly as possible.

Technical Preliminaries As we showed in previous work [14], syntactic restrictions can be conveniently represented as a set of *(algebraic) datatypes*, for which some SMT solvers have dedicated decision procedures [7, 13]. For instance, given a function $f : (x : \mathsf{Int}) \times (y : \mathsf{Int}) \to \mathsf{Int}$ and the context-free grammar R below specifying what integer (I) and Boolean (B) terms can appear in candidate solutions for f:

$$I ::= 0 \mid 1 \mid x \mid y \mid I + I \mid I - I \mid \mathsf{ite}(B, I, I) \tag{1}$$

$$B ::= B \geqslant B \mid I \approx I \mid \neg B \mid B \wedge B \tag{2}$$

our SyGuS solver generates the following mutually recursive datatypes:

$$\mathcal{I} = \mathsf{0} \mid \mathsf{1} \mid \mathsf{x} \mid \mathsf{y} \mid \mathsf{plus}(\mathcal{I}, \mathcal{I}) \mid \mathsf{minus}(\mathcal{I}, \mathcal{I}) \mid \mathsf{ite}(\mathcal{B}, \mathcal{I}, \mathcal{I}) \tag{3}$$

$$\mathcal{B} = \mathsf{geq}(\mathcal{I}, \mathcal{I}) \mid \mathsf{eq}(\mathcal{I}, \mathcal{I}) \mid \mathsf{not}(\mathcal{B}) \mid \mathsf{and}(\mathcal{B}, \mathcal{B}) \tag{4}$$

Each datatype constructor corresponds to a production rule of R, e.g. plus corresponds to the rule $I ::= I + I$. A datatype term such as $\mathsf{plus}(\mathsf{x}, \mathsf{y})$ represents the arithmetic term $x + y$. We will use these datatypes as a running example.

For a datatype term t, we write $\mathsf{is}_{\mathsf{C}}(t)$ to denote the *discriminator* predicate that is satisfied exactly when t is interpreted as a datatype whose top constructor is C. We write $\mathsf{sel}_n^{\tau}(t)$ to denote a *shared selector* [15] applied to t, interpreted as the n^{th} child of t with type τ if one exists, and interpreted as an arbitrary element of τ otherwise. A term consisting of zero or more consecutive nested applications of shared selectors applied to a term t is a *shared selector chain (for t)*.

2 Smart Enumerative SyGuS

Our *smart enumerative SyGuS* approach CVC4SY_S, is based on finding solutions for an evolving set of constraints in an extension of the quantifier-free fragment of algebraic datatypes. These constraints are constructed to rule out many redundant solutions while not overconstraining the problem, potentially missing actual solutions.

In detail, candidate solutions for the function $f : \tau_1 \to \tau_2$ to be synthesized are constructed by maintaining a set of constraints F, initially empty, for a first-order variable d ranging over the datatype representing τ_2. For example, consider again the function f with the syntactic restrictions expressed by the datatypes in Equations 3 and 4. If the term generator finds a model for F, it provides to the solution verifier the integer term which corresponds to the value of d in the model; for example, it provides $x + 1$ when d is interpreted as $\mathsf{plus}(\mathsf{x}, 1)$. In turn, if the solution verifier finds that $x + 1$ is not a solution, it provides the *blocking constraint* $\neg\mathsf{is}_{\mathsf{plus}}(d) \vee \neg\mathsf{is}_{\mathsf{x}}(\mathsf{sel}_1^{\mathcal{I}}(d)) \vee \neg\mathsf{is}_{\mathsf{1}}(\mathsf{sel}_2^{\mathcal{I}}(d))$, i.e., the datatype constraint that rules out the current value for d, which is then added to F. This is a *syntactic* constraint on future candidate solutions from the term generator. Its atoms are discriminators applied to shared selector chains.

CVC4SY_S uses a number of optimization techniques in addition to the basic loop above, which we describe in the remainder of this section. These techniques produce blocking constraints via the lemmas-on-demand paradigm [6] that eagerly rule out spurious candidates, *prior* to the solution verification step. Additionally, whenever possible, it *strengthens* blocking constraints via novel generalization techniques, with the effect of ruling out larger classes of candidates.

Blocking via Theory Rewriting with Structural Generalization As we describe in previous work [14], the enumerative solver of CVC4 uses its rewriter as an oracle for discovering when candidate solutions are redundant. The motivation is that for any two equivalent terms t and s, only one of them needs to be checked with the solution verifier, since either both t and s are solutions to the synthesis conjecture or neither is. Given a term t, we write $t{\downarrow}$ to denote its *rewritten form*. Note that it is possible for equivalent terms not to have the same rewritten form. This is a consequence of the trade-offs in the implementation of CVC4's rewriter, which must balance efficiency and completeness.

As an example, suppose that the term enumerator previously generated $x+y$ and that d's current value is the datatype term representing $y + x$, where, however, $(x + y){\downarrow} = (y + x){\downarrow}$. We first generate a blocking constraint template $R[z]$ of the form $\neg\mathsf{is}_{\mathsf{plus}}(z) \vee$

$\neg \mathsf{is}_{\mathsf{y}}(\mathsf{sel}_1^{\mathcal{I}}(z)) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_2^{\mathcal{I}}(z))$, where z is a fresh variable. This template is subsequently instantiated with $z \mapsto u$ for any shared selector chain u of type $\mathcal{I}$ that currently (or later) appears in F, starting with d itself. This has the effect of ruling out all candidate solutions that have $y + x$ as a subterm, which is justified by the fact that each such term is equivalent to one in which all occurrences of $y + x$ are replaced by $x + y$.

We employ a refinement of this technique, which we call *theory rewriting with structural generalization*, which searches for and then blocks only the minimal skeleton of the term under test that is sufficient for determining its rewritten form. For example, consider the if-then-else term $t = \mathsf{ite}(x \approx 0 \wedge y \geqslant 0,\ 0,\ x)$, This term is equivalent to x, regardless of the value of predicate $y \geqslant 0$. This can be confirmed by the rewriter by computing that $\mathsf{ite}(x \approx 0 \wedge w,\ 0,\ x)\downarrow = x$ where w is a fresh Boolean variable. Then, instead of generating a constraint that blocks only (the datatype value corresponding to) t, we generate a stronger constraint that does not depend on the subterm $y \geqslant 0$. In other words, this blocking constraint rules out all candidate solutions that contain the subterm $\mathsf{ite}(x \approx 0 \wedge w,\ 0,\ x)$, for *any* term w. We compute these generalizations using a recursive algorithm that iteratively replaces *each* subterm of the current candidate with a fresh variable, and checks whether its rewritten form remains the same.

Blocking via CEGIS with Structural Generalization Synthesis solvers based on CEGIS maintain a list of *refinement points* that witness the infeasibility of previous candidate solutions. That is, given a synthesis conjecture $\exists f.\, \forall \bar{x}.\, \varphi[f,\, \bar{x}]$, the solver maintains a growing list $\bar{p}_1, \ldots, \bar{p}_n$ of values for $\bar{x}$ that witness the infeasibility of previous candidates $u_1, \ldots, u_n$ for f. Then, when a new candidate u is generated, we first check whether $\varphi[u,\, \bar{p}_i]$ is false for some $i \leqslant n$. When a candidate u fails to satisfy $\varphi[u, \bar{p}_i]$, CVC4SY_S further applies a form of generalization analogous to the structural generalization described above. We call this *CEGIS with structural generalization*, where the goal is to find the minimal skeleton of u that also fails to satisfy some refinement point.

For example, suppose f is the function to synthesize, φ includes the constraint $f(x, y) \leqslant x - 1$, and $p_1 = (3, 3)$ is a refinement point. Then, the candidate term $u[x, y] = \mathsf{ite}(x \geqslant 0,\ x,\ y + 1)$ will be discarded, because $\mathsf{ite}(3 \geqslant 0,\ 3,\ 4) \not\leqslant 2$. Notice, however, that *any* candidate $u' = \mathsf{ite}(x \geqslant 0,\ x,\ w)$ is falsified by p_1, regardless of what w is, since $u'[3, 3] \leqslant 2$ is equivalent to $3 \leqslant 2$. This indicates that we can block *all* ite candidate terms with condition $x \geqslant 0$ and true branch x. We can express this constraint in CVC4SY_S by dropping the disjuncts that relate to the false branch of the ite term. This form of blocking is particularly useful when synthesizing multiple functions $(f_1,$ $\ldots, f_n)$, since it is often the case that a candidate for a single f_i is already sufficient to falsify the specification, regardless of what the candidates for the other functions are.

Evaluation Unfolding This technique uses *evaluation functions* to encode the relationship between the datatype terms assigned to d and their analogs in the theory T. For example, the evaluation function for the datatype $\mathcal{I}$ defined in (3) is a function $\mathsf{E}_{\mathcal{I}} : \mathcal{I} \times \mathsf{Int} \times \mathsf{Int} \mapsto \mathsf{Int}$ defined axiomatically so that $\mathsf{E}_{\mathcal{I}}(d, m, n)$ denotes the result of evaluating d by interpreting any occurrences of x and y in d respectively as m and n and interpreting the other constructors as the corresponding arithmetic/Boolean operators, e.g. $\mathsf{E}_{\mathcal{I}}(\mathsf{minus}(\mathsf{x}, \mathsf{y}), 5, 3)$ is interpreted as 2. When a refinement point $\bar{c}$ is generated, we add a constraint requiring that the evaluation of d at $\bar{c}$ must satisfy the specification. For

example, for conjecture $\exists f.\, \forall x.\, f(x+1, x) \leqslant 0$, and refinement point $x \mapsto 1$, we add the constraint $\mathsf{E}_{\mathcal{I}}(d, 2, 1) \leqslant 0$. Then, when a literal $\mathsf{is}_{\mathsf{C}}(t)$ is asserted for a term t of type $\mathcal{I}$, we can add a constraint corresponding to the one-step unfolding of the evaluation of t. Specifically, when $\mathsf{is}_{\mathsf{ite}}(d)$ is asserted, we generate the constraint

$$\mathsf{is}_{\mathsf{ite}}(d) \Rightarrow \mathsf{E}_{\mathcal{I}}(d, 2, 1) \approx \mathsf{ite}(\mathsf{E}_{\mathcal{B}}(\mathsf{sel}_1^{\mathcal{B}}(d), 2, 1), \mathsf{E}_{\mathcal{I}}(\mathsf{sel}_1^{\mathcal{I}}(d), 2, 1), \mathsf{E}_{\mathcal{I}}(\mathsf{sel}_2^{\mathcal{I}}(d), 2, 1))$$

indicating that the evaluation of d on point $(2, 1)$ indeed behaves like an ite term when d has top symbol ite. Our implementation adds these constraints for all terms t whose top symbols correspond to ite or Boolean connectives. For terms t whose top symbol is any of the other operators, we add constraints corresponding to their total evaluation of t when the value of t is fully determined, for example, $t \approx \mathsf{plus}(\mathsf{x}, \mathsf{y}) \Rightarrow \mathsf{E}_{\mathcal{I}}(t, 2, 1) \approx 3$. Notice this constraint with $t = d$ along with the refinement constraint $\mathsf{E}_{\mathcal{I}}(d, 2, 1) \leqslant 0$ suffices to show that d cannot be $\mathsf{plus}(\mathsf{x}, \mathsf{y})$.

3 Fast Enumerative SyGuS

The techniques in the previous section prune the search space so that often, only a small subset of the entire possible set of terms is considered for a given term size bound. The main bottleneck, however, is managing the large number of blocking constraints generated. Moreover, the benefits of this approach are limited when the grammar or specification does not admit opportunities for generalization.

For this reason, we have also developed CVC4SY_F, which, in the spirit of other SyGuS solvers (notably ESOLVER [17]), relies on a principled brute-force approach for term generation. In contrast to other solvers, however, which are built as layers on top of the core SMT reasoner, CVC4SY_F is fully integrated as a subsolver of CVC4, so communication with other components has almost no overhead. This technique, *fast enumerative synthesis*, does not use constraint solving to generate new terms. As a result, the majority of optimizations from Section 2 are incompatible with it.

Algorithm To generate terms up to a given size k, we maintain a set S_τ^k of terms of type τ and size k for each datatype τ corresponding to a non-terminal symbol of our input grammar R. First, we compute for each such τ the set $\mathcal{C}_\tau$ of its *constructor classes*, an equivalence relation over the constructors of τ that groups them by their type. For example, the constructor classes for $\mathcal{I}$ are $\{\mathsf{x}, \mathsf{y}, 0, 1\}$, $\{\mathsf{plus}, \mathsf{minus}\}$ and $\{\mathsf{ite}\}$. Then, we use the following procedure for generating all terms of size k for type τ:

FASTENUM(τ, k):
For all:
- Constructor classes $C \in \mathcal{C}_\tau$, whose elements have type $\tau_1 \times \ldots \times \tau_n \to \tau$,
- Tuple of naturals $(k_1, \ldots k_n)$ such that $k_1 + \ldots + k_n + \mathsf{ite}(n > 0, 1, 0) = k$,

(a) Run FASTENUM(τ_i, k_i) for each $i = 1, \ldots, n$,
(b) Add $\mathsf{C}(t_1, \ldots, t_n)$ to S_τ^k for all tuples $(t_1, \ldots, t_n)$ with $t_i \in S_{\tau_i}^{k_i}$ and all constructors $\mathsf{C} \in C$.

The recursive procedure FASTENUM(τ, k) populates the set S_τ^k of all terms of type τ with size k. These sets are cached globally. We incorporate an optimization that only

adds terms $\mathsf{C}(t_1, \ldots, t_n)$ to S^k_τ whose corresponding terms in the theory T are unique up to rewriting. This mimics the effect of blocking via theory rewriting as described in Section 2. For example, $\mathsf{plus}(\mathsf{y}, \mathsf{x})$ is not added to $S^1_\mathcal{I}$ if that set already contains $\mathsf{plus}(\mathsf{x}, \mathsf{y})$, noting that $(x + y){\downarrow} = (y + x){\downarrow}$. By construction of S^k_τ for $k \geqslant 1$, this has the cascading effect of excluding all terms having $y + x$ as a subterm.

We observe that theory rewriting with structural generalization cannot be easily incorporated into this scheme since it requires the use of a constraint solver, something that the above algorithm seeks to avoid.

4 Hybrid Approach: Variable-Agnostic Enumerative SyGuS

We follow a third approach, in solver CVC4SY_H, that combines elements of the previous approaches. The idea is to use the (smart) approach from Section 2 to generate terms, but then generate *multiple* candidate solutions from each term using a fast subprocedure we call a *concretizer*. We implement an instance of this scheme, which we call *variable-agnostic* term generation, that produces only terms that are unique modulo alpha-equivalence. In our running example, when a term t such as $x + 1$ is produced, the concretizer produces all terms generated by the grammar R that are alpha-equivalent to t, namely, $\{x + 1, y + 1\}$ in this case. The advantage of this approach is that CVC4SY_H can block any term whose variables are not canonically ordered; that is, assuming for instance that $x \prec y$, it may block terms like $1 - y$ and $y + y$, noting they are alpha-equivalent to $1 - x$ and $x + x$, respectively. To implement this blocking scheme, we introduce unary Boolean predicates pre_x and $post_x$ for each variable x in our grammar, where pre_x (resp., $post_x$) holds for t if and only if variable x occurs in a depth-first left-to-right traversal of our candidate term before (resp., after) traversing to the position indicated by the selector chain t. We encode the semantics of these predicates based on the arguments of constructors in our signature, e.g. $\mathsf{is}_{\mathsf{plus}}(z) \Rightarrow (pre_x(z) \approx pre_x(\mathsf{sel}^\mathcal{I}_1(z)) \wedge post_x(\mathsf{sel}^\mathcal{I}_2(z)) \approx post_x(z))$. We then assert that pre_x and pre_y are false for our top-level variable d, and require $\mathsf{is}_{\mathsf{y}}(z) \Rightarrow pre_x(z)$ for all z, stating that x must come before y in the traversal of any generated term.

This technique is useful for grammars with many variables, such as grammars in invariant synthesis problems, where the number of terms of small size is prohibitively large. Blocking based on theory rewriting (with generalization) from Section 2 is compatible with this technique and is used in CVC4SY_H. However, the other optimizations are disabled, since they prune solutions in a way that is not agnostic to variables.

5 Evaluation

We evaluated the above techniques in CVC4SY on four benchmark sets: invariant synthesis benchmarks from the verification of Lustre [11] models; a set from work on synthesizing invertibility conditions for bit-vector operators [12] (IC-BV); a set of bit-vector invariant synthesis problems [2] (CegisT); and the SyGuS-COMP 2018 [1] benchmarks from five tracks: assorted problems (General), conditional linear arithmetic problems (CLIA), invariant synthesis problems (INV), and programming-by-examples problems [10] with a set over bit-vectors (PBE-BV) and another over strings (PBE-Str). We

Set	#	**a+si**	**a**	**s**	**s-cg**	**s-eu**	**s-rg**	**s-r**	**f**	**f-r**	**h**	**h-rg**	**h-r**	EUS
General	413	**293**	237	228	229	232	230	220	237	226	221	225	213	290
Gen-CrCi	214	**159**	**159**	**159**	**159**	143	**159**	**159**	155	132	130	137	125	152
CLIA	88	**86**	20	20	19	19	19	18	20	16	16	16	16	85
INV	127	109	109	109	109	109	109	109	**110**	109	109	109	109	68
PBE-BV	753	**751**	**751**	721	721	721	721	628	**751**	717	721	721	628	745
PBE-Str	109	**105**	**105**	104	104	104	87	75	**105**	103	102	87	75	74
Subtotal	1704	**1503**	1381	1341	1341	1328	1325	1209	1378	1303	1299	1295	1166	1414
IC-BV	160	135	135	135	132	130	130	133	**138**	132	128	126	127	
CegisT	79	**56**	43	43	43	43	42	41	42	42	42	42	41	
Lustre	485	**255**	**255**	**255**	**255**	218	211	221	231	213	248	244	234	
Total	2428	**1949**	1814	1774	1771	1719	1708	1604	1789	1690	1717	1707	1568	

Table 1. Summary of number of problems solved per benchmark set. Best results are in **bold**.

also considered separately the CrCi subset from General, which corresponds to cryptographic circuit synthesis. We ran our experiments on a cluster equipped with Intel E5-2637 v4 CPUs running Ubuntu 16.04, providing one core, 1800 seconds, and 8GB RAM for each job. Results are summarized in Table 1 and Figure 2. We denote the strategies from Sections 2, 3, and 4 by **s**, **f** and **h**, respectively (smart, fast, and hybrid); disabling the optimizations from Section 2 is marked by "-" and the suffixes **r** (rewriting), **rg** (rewriting with structural generalization), **cg** (CEGIS with structural generalization), and **eu** (evaluation unfolding). We also evaluated two meta-strategies of CVC4SY: **a** and **a+si**. The auto strategy **a** picks a strategy based on the properties of the problem: **f** for PBE problems and for problems without the Boolean type or the ite operator in their grammar and **s** otherwise. Strategy **a+si** uses the single-invocation solver [14] on problems that are amenable to quantifier elimination and **a** otherwise. We use the state-of-the-art SyGuS solver EUSOLVER [4] (**EUS**) as a baseline, but only for SyGuS-COMP benchmarks due to limitations in its parser.

Overall, strategy **s** excels on more challenging benchmark sets such as Lustre and Gen-Crci, while strategy **f** excels on the majority of the others. The gains for **f** are especially significant on PBE problems, where it outperforms both **s** and **EUS** by several orders of magnitude. Such gains are significant given that CVC4 won this track at SyGuS-COMP 2018 by employing **s** alone, and a variant of **EUS** won it in 2017. This result can be explained as a consequence of two factors. First, the string and bit-vector grammars contain many operators with the same type, making the constructor class optimization of the **f** algorithm very effective. Second, although not described in this paper, all solvers in our evaluation use divide-and-conquer algorithms for PBE problems [4], which are not compatible with the optimizations **cg** and **eu**. The most important optimization for all CVC4SY strategies and with all benchmark sets is **r**. The optimization **eu** is especially effective when grammars contain ite and Boolean connectives, such as those in the Lustre set and in some subsets of General, on which we can

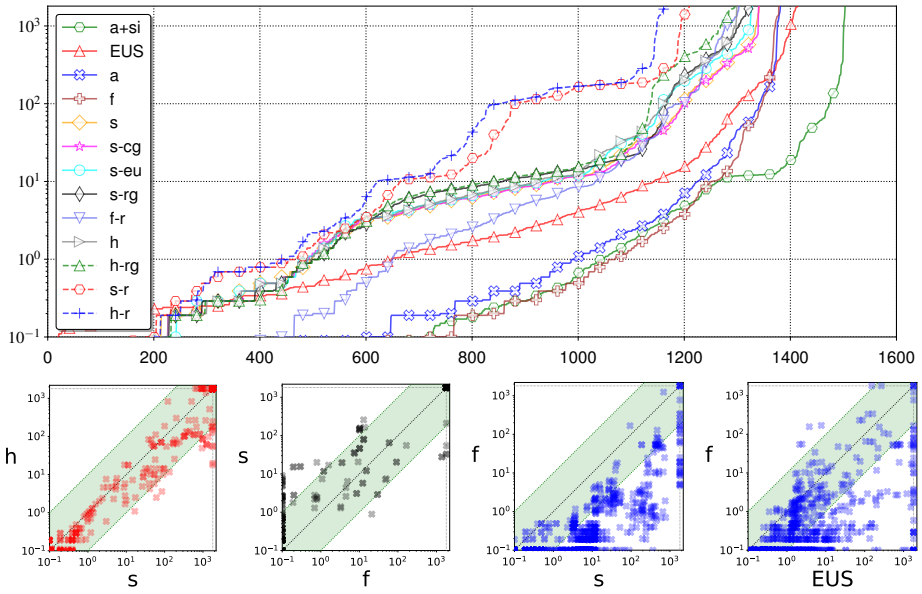

Fig. 2. Cactus plot on commonly supported benchmark sets. The first scatter plot is for the Lustre set, the second for the Gen-Crci set, and the latter two for the 862 benchmarks from the PBE sets.

see the biggest gains of **s** with respect to **s-eu**; **cg** is more helpful for IC-BV, with a few harder benchmarks only solved due to this technique.

The first scatter plot in Figure 2 shows the advantage of **h** over **s** on Lustre, a benchmark set containing invariant synthesis problems with dozens of variables. We remark this configuration excels at quickly finding small solutions for problems with many variables, although solves fewer problems overall. The second scatter plot shows that while **s** takes significantly longer on easy problems, it outperforms **f** in the long run. The last two plots show that **f** significantly outperforms the state of the art on PBE benchmarks.

For all benchmark sets, the auto strategy **a** chooses the best enumerative strategy of CVC4SY with only a few exceptions, and hence it is the default configuration of CVC4SY. Due to specialized synthesis techniques [14, 4], both **a+si** and **EUS** outperform the purely enumerative strategies of CVC4. This is reflected in the cactus plot on the commonly supported benchmark sets, where **a** and **f** solve more benchmarks than **EUS** for lower times but then **EUS** solves more benchmarks in the end. For **a+si**, the cactus plot shows that it outperforms **EUS** significantly. Nevertheless, we remark that **a+si** is able to solve only 393 (16%) of the overall benchmarks using only single invocation techniques. Hence, we conclude that both smart and fast enumerative strategies are critical subcomponents in our approach to syntax-guided synthesis.

Acknowledgments This work was partially supported by the National Science Foundation under award 1656926 and by the Defense Advanced Research Projects Agency under award FA8650-18-2-7854.

References

[1] SyGuS-COMP 2018. `http://sygus.seas.upenn.edu/SyGuS-COMP2018.html`, 2018.

[2] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. Counterexample guided inductive synthesis modulo theories. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification (CAV), Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2018.

[3] 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 M. Irlbeck, D. A. Peled, and A. Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

[4] 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 (TACAS)*, pages 319–336, 2017.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177. Springer-Verlag, 2011.

[6] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in sat modulo theories. In M. Hermann and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer Berlin Heidelberg, 2006.

[7] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electr. Notes Theor. Comput. Sci.*, 174(8):23–37, 2007.

[8] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.

[9] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[10] S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Computer Science*, pages 9–14. Springer, 2016.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[12] A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli. Solving quantified bit-vectors using invertibility conditions. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification (CAV), Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 236–255. Springer, 2018.

[13] A. Reynolds and J. C. Blanchette. A decision procedure for (co) datatypes in SMT solvers. In *International Conference on Automated Deduction*, pages 197–213. Springer International Publishing, 2015.

[14] 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*, pages 198–216, 2015.

[15] A. Reynolds, A. Viswanathan, H. Barbosa, C. Tinelli, and C. Barrett. Datatypes with shared selectors. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, pages 591–608, 2018.

[16] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. pages 404–415. ACM, 2006.

[17] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296, 2013.