SyGuS Techniques in the Core of an SMT Solver

Andrew Reynolds
University of Iowa
Iowa City, Iowa, USA
andrew.j.reynolds@gmail.com

Cesare Tinelli
University of Iowa
Iowa City, Iowa, USA
tinelli@uiowa.edu

We give an overview of recent techniques for implementing syntax-guided synthesis (SyGuS) algorithms in the core of Satisfiability Modulo Theories (SMT) solvers. We define several classes of synthesis conjectures and corresponding techniques that can be used when dealing with each class of conjecture.

1 Introduction

A synthesis conjecture asks whether there exists a structure for which some property holds universally. Traditionally, such conjectures are very challenging for automated reasoners. Syntax-guided synthesis (or SyGuS) is a recently introduced paradigm [1] where a user may provide syntactic hints to guide an automated reasoner in its search for solutions to synthesis conjectures. A number of recent solvers based on this paradigm have been successfully used in applications, including correct-by-construction program snippets [33] and for implementation of distributed protocols [36].

Satisfiability Modulo Theories (SMT) solvers have historically been used as subroutines for automated synthesis tasks [18, 35, 4, 37]. More recently, we have advocated in Reynolds et al. [29] for SMT solvers to play a more active role in solving synthesis conjectures, including being used as stand-alone tools. In particular, we have recently instrumented the SMT solver CVC4 [6] with new capabilities which make it efficient for synthesis conjectures, and entered it in the past several editions of the syntax-guided synthesis competition [2, 3] where it placed first in a number of categories. This work has shown that SMT solvers can be powerful tools for handling synthesis conjectures using two orthogonal techniques:

1. **Synthesis via Quantifier Instantiation** The first leverages the support in SMT solvers for first-order quantifier instantiation, which has been used successfully in a number of approaches for automated theorem proving [16, 24, 30]. Quantifier-instantiation techniques developed for SMT can be extended and used as a complete procedure for certain classes of synthesis conjectures.
2. **Synthesis via Syntax-Guided Enumeration** The second technique follows the syntax-guided synthesis paradigm [1]. More specifically, it simulates an enumerative search strategy in the core of the SMT solver by leveraging its native support for algebraic datatypes.

This paper gives an overview of the way these synthesis techniques can be embedded in the core of SMT solvers based on the DPLL(T) framework [25]. We focus on recent advances for achieving efficiency while giving consideration to the quality of solutions generated using both these techniques.

Overview In Section 2, we give preliminary definitions and introduce several classes of synthesis conjectures. In Section 3, we introduce the concept of refutation-based synthesis and summarize the scenarios in which it can be applied to synthesis conjectures. In Section 4, we describe a synthesis technique based on quantifier instantiation, its properties, and associated challenges. In Section 5, we describe current work on developing syntax-guided synthesis techniques in the core of an SMT solver. In Section 6, we conclude with several directions for future work.

Submitted to:
SYNT 2017

2 Preliminaries

We consider synthesis in the context of a background theory T in many-sorted logic. Formally, a theory is a pair $(\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, the intended models for T. A formula is *T-satisfiable* (respectively, *T-valid*, *T-unsatisfiable*) if it is satisfied by some (respectively, every, no) interpretation in $\mathbf{I}$. A *synthesis conjecture* is a formula of the form $\exists \vec{f} \forall \vec{x} P[\vec{f}, \vec{x}]$ with $\vec{f} = (f_1, \ldots, f_n)$ and $\vec{x} = (x_1, \ldots, x_k)$, where each f_i in $\vec{f}$ has type of the form $\tau_1 \times \ldots \times \tau_{n_i} \to \tau$, where $\tau_1, \ldots, \tau_{n_i}, \tau$ are (first-order) sorts in Σ and P is a *first-order* Σ-formula, which means that in P the second-order variables $\vec{f}$ are fully applied to arguments. Here, we write $P[\vec{f}, \vec{x}]$ to denote that the free variables of formula P are a subset of those in tuples $\vec{f}$ and $\vec{x}$, and use this convention throughout the paper. A *solution* to the synthesis conjecture is a substitution of the form $\{f_1 \mapsto \lambda \vec{y}_1 t_1, \ldots, f_n \mapsto \lambda \vec{y}_n t_n\}$ where $\vec{y}_1, \ldots, \vec{y}_n$ are bound variables and $t_1, \ldots, t_n$ are Σ-terms with no second-order variables[1] such that $\forall \vec{x} P[(\lambda \vec{y}_1 t_1, \ldots, \lambda \vec{y}_n t_n), \vec{x}]$, after β-reduction, is T-valid in T. Note that we restrict our consideration to solutions were none of the f_i need to be defined recursively.

Optionally, we may be interested in synthesis conjectures in the presence of *syntactic restrictions*. We specify syntactic restrictions by a grammar $\mathscr{R} = (s_0, S, R)$ where s_0 is an initial symbol, S is a set of symbols with $s_0 \in S$, and R is a set of rules of the form $s \to t$, where $s \in S$ and t is a term built from the symbols in the signature of theory T, free variables, and symbols from S. The rules define a rewrite relation over such terms, also denoted by $\to$, as expected. We say a term t is *generated* by $\mathscr{R}$ if $s_0 \to^* t$ where $\to^*$ is the reflexive-transitive closure of $\to$ and t does not contain symbols from S. For example, the terms x, $(x+x)$ and $((1+x)+1)$ are all generated by the grammar $\mathscr{R} = (\mathsf{I}, \{\mathsf{I}\}, \{\mathsf{I} \to x, \mathsf{I} \to 1, \mathsf{I} \to (\mathsf{I}+\mathsf{I})\})$. We will write a grammar like this in BNF-style as:

$$\mathsf{I} \quad \to \quad x \mid 1 \mid (\mathsf{I}+\mathsf{I})$$

We say that a solution $\lambda \vec{x} t$ meets the syntactic restrictions of $\mathscr{R}$ if t is generated by it.

2.1 Classes of Synthesis Conjectures

We describe SMT approaches for synthesis specialized to particular classes of conjectures. We introduce the following terminology for defining those classes with respect to a given Σ-theory T.

Definition 1 (Input-Output Example Conjectures)**.** *An input-output example synthesis conjecture is a formula of the form:*

$$\exists \vec{f} \forall \vec{x} (\bigwedge_{k=0}^{n} \vec{x} \approx \vec{i}_k \Rightarrow \vec{f}(\vec{x}) \approx \vec{o}_k)$$

where for each $k = 1, \ldots, n$, $\vec{i}_k$ and $\vec{o}_k$ are tuples of constants from Σ.

Example 1. If T is the theory of integer arithmetic with the usual signature, the formula:

$$\exists f \forall x (x \approx 1 \Rightarrow f(x) \approx 2) \wedge (x \approx 2 \Rightarrow f(x) \approx 3) \wedge (x \approx 7 \Rightarrow f(x) \approx 8)$$

is an input-output example conjecture. □

[1] In particular, it contains no variables from $\vec{f}$.

Syntax \ Conjecture	I/O Examples	Single Invocation	Non Single Invocation
restricted	Enumerative + I/O sym breaking	Enumerative, or CEGQI+reconstruction	Enumerative
unrestricted	CEGQI (trivially)	CEGQI	Enumerative (using default restrctions)

Figure 1: Refutation-based techniques used by SMT solvers for solving classes of synthesis conjectures.

Definition 2 (Single invocation Conjectures)**.** *A single invocation conjecture is a formula of the form:*

$$\exists \vec{f} \forall \vec{x} P[\vec{f}(\vec{x}), \vec{x}]$$

where $P[\vec{f}(\vec{x}), \vec{x}]$ is an instance of a formula $P[\vec{y}, \vec{x}]$ that does not contain any second-order variables. In other words, single invocation conjectures are those where all functions in $\vec{f}$ are applied to the same argument tuple $\vec{x}$.

Notice that input-output example conjectures are a subset of single invocation conjectures.

Example 2. The formula:

$$\exists f \forall x y \, (f(x,y) \geq x \wedge f(x,y) \geq y)$$

stating that f returns a value greater than its arguments is a single invocation synthesis conjecture. □

All other synthesis conjectures that do not meet the criteria of the above definition we refer to as *non-single invocation* conjectures.

Example 3. The formula:

$$\exists f \forall x y \, (f(x,y) \approx f(y,x)) \tag{1}$$

stating that f is a commutative function is a non-single invocation synthesis conjecture. □

3 Refutation-Based Synthesis

We use a *refutation-based approach* for synthesis in SMT solvers [31] that takes as input the negation of a synthesis conjecture $\neg \exists \vec{f} \forall \vec{x} P[\vec{f}, \vec{x}]$, which is equivalent to $\forall \vec{f} \exists \vec{x} \neg P[\vec{f}, \vec{x}]$. In this approach, a solution for $\vec{f}$ can be extracted from a proof of unsatisfiability[2], whereas a satisfiable response from the solver indicates that the synthesis conjecture has no solutions. This paper will focus solely on the former case, that is, we consider only synthesis conjectures that have solutions.

Figure 1 summarizes the refutation-based techniques we use in DPLL(T)-based SMT solvers for handling various classes of synthesis conjectures, both with and without syntactic restrictions. We give details on variants of counterexample-guided quantifier instantiation (CEGQI) and syntax-guided enumerative search in the remainder of the paper. As indicated in the figure, counterexample-guided quantifier instantiation (Section 4) is applicable to single invocation conjectures only. It also trivially applies to input-output examples without syntactic restrictions, which we describe in Section 4.2. It typically is used only when no syntactic restrictions are associated with the conjecture, although Section 4.3 describes a technique for reconstructing solutions from counterexample-guided instantiation that satisfy

[2] We will show examples of how solutions are extracted later in the paper.

syntactic restrictions. For other classes of conjectures, we use syntax-guided enumerative search (Section 5). In the case where a synthesis conjecture is not single invocation but has no syntactic restrictions, we use a set of *default restrictions*, that is, those that allow all constructable Σ-terms as solutions. For input-output examples, we may use a technique for breaking symmetries in the search space based on evaluating input-output examples which we describe in Section 5.2.

4 Synthesis via Counterexample-Guided Quantifier Instantiation

In previous work [29], we developed an efficient technique for single invocation synthesis conjectures without syntactic restrictions. In this section, we give a brief review of this technique and mention current challenges associated with this approach. Unless otherwise stated, in all examples we will use linear integer arithmetic as the background theory T with sort Int for the set of all integers. We will write $t_1 > t_2 > t_3$ as an abbreviation for $t_1 > t_2 \wedge t_2 > t_3$.

Example 4. Consider the synthesis conjecture:

$$\exists f \forall x y\, (x > y+1 \Rightarrow x > f(x,y) > y) \wedge (y > x+1 \Rightarrow y > f(x,y) > x) \tag{2}$$

where f is of type $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ and all other variables are of type Int. This conjecture is single invocation and states that f is a function that, under certain conditions on the inputs x and y, returns a value strictly between those inputs. As noted by Reynolds et al. [29], the second-order formula above is equivalent to the first-order formula:

$$\forall x y \exists z\, (x > y+1 \Rightarrow x > z > y) \wedge (y > x+1 \Rightarrow y > z > x) \tag{3}$$

where z is of type Int. This formula states that for every two values x and y satisfying certain restrictions, there exists a "return" value z that is strictly between those values.

In contrast to formula (2), formula (3) can be processed with SMT techniques for first-order quantified linear arithmetic [8, 9, 12, 13, 30]. In particular, since these techniques are refutation-based, we consider the negation of (3):

$$\exists x y \forall z\, \neg((x > y+1 \Rightarrow x > z > y) \wedge (y > x+1 \Rightarrow y > z > x)) \tag{4}$$

Using quantifier instantiation, we can show that the instances of the innermost quantified formula:

$$\neg((x > y+1 \Rightarrow x > x+1 > y) \wedge (y > x+1 \Rightarrow y > x+1 > x)) \wedge$$
$$\neg((x > y+1 \Rightarrow x > y+1 > y) \wedge (y > x+1 \Rightarrow y > y+1 > x))$$

which simplify to $x > y+1$ and $y > x+1$ respectively, are together T-unsatisfiable. As described in [29], a solution for f in (2) can be extracted from the instantiations required for showing (4) T-unsatisfiable. In particular, we construct a conditional function whose return values are the terms we considered as instances of the negated first-order conjecture. In this case, from the above instances of (4) we construct the function

$$\lambda x y\, \mathsf{ite}((x > y+1 \Rightarrow x > x+1 > y) \wedge (y > x+1 \Rightarrow y > x+1 > x), x+1, y+1)$$

which states that when the conjecture holds for $x+1$, return $x+1$, otherwise return $y+1$. After simplification, this gives the solution $f = \lambda x y\, \mathsf{ite}(x \leq y+1, x+1, y+1)$, which indeed is a solution for our original conjecture in (2). □

The key technical challenge in the previous example was to determine a T-unsatisfiable set of instances of (4). We use a technique called *counterexample-guided quantifier instantiation* (CEGQI) for determining these instances. Variants of CEGQI have been used in a number of recent works [23, 19, 9, 12, 14, 30]. A detailed description of the technique can be found in [30]; we summarize the most important details in the following.

For a negated single invocation synthesis conjecture $\neg\exists\vec{f}\forall\vec{x}P[\vec{f}(\vec{x}),\vec{x}]$, our goal is to find a set Γ of instances of the innermost body of the equivalent first-order formula $\exists\vec{x}\forall\vec{z}\neg P[\vec{z},\vec{x}]$ that are collectively T-unsatisfiable. We incrementally construct the set $\Gamma = \{\neg P[\vec{t}_1,\vec{x}],\neg P[\vec{t}_2,\vec{x}],\ldots\}$ where each $\vec{t}_i$ is chosen by a *selection function*.

Definition 3 (Selection Function)**.** *A* selection function *for* $\forall\vec{z}\neg P[\vec{z},\vec{x}]$ *takes as input:*

1. *a* Σ*-interpretation* $\mathscr{I}$*,*
2. *a tuple of fresh variables* $\vec{k}$ *with the same length and type as* $\vec{z}$*, and*
3. *a set of formulas* $\Gamma = \{\neg P[\vec{t}_1,\vec{x}],\ldots,\neg P[\vec{t}_n,\vec{x}],P[\vec{k},\vec{x}]\}$ *where* $\mathscr{I}\models\Gamma$*.*

It returns a tuple of terms $\vec{t}$ *of the same type as* $\vec{k}$ *whose free variables are a subset of* $\vec{k}$*.*

We use a selection function for finding the next instance $\neg P[\vec{t},\vec{x}]$ of $\neg P[\vec{z},\vec{x}]$ to add to Γ. Typically, the terms $\vec{t}$ are chosen based on the model $\mathscr{I}$ for Γ. In particular, a model-based selection function is one that choses $\vec{t}$ based on the value of $\vec{k}$ in $\mathscr{I}$. With sufficient conditions on the selection function used, one may develop sound and complete instantiation procedures for the satisfiability of $\exists\forall$-formulas in certain theories, which hence can be used as sound and complete procedures for single invocation synthesis conjectures without syntactic restrictions.

Selection functions are specific to the background theory T and are often inspired by quantifier elimination procedures. Consider the case of a formula $\forall z\neg P[z,\vec{x}]$ with a single quantified variable (corresponding to an original synthesis conjecture involving a single function). A selection function will chose instantiations for z based on models $\mathscr{I}$ that satisfy $P[k,\vec{x}]$. For linear real arithmetic, a selection function for this formula may choose to return a tuple of terms corresponding to the maximal lower bound (respectively minimal upper bound) for k in $\mathscr{I}$, where virtual terms such as ∞ and δ may be necessary. This is analogous to the quantifier elimination procedure by Loos and Weispfenning [22]. We may also add the midpoint of the maximal lower and minimal upper bounds for k, analogous to Ferrante and Rackoff's method [15]. It is also possible to devise selection functions for linear integer arithmetic that take into account implied divisibility constraints for k, similar to Cooper's method [11].

We remark that there is a correspondence between three classes of procedures:

1. quantifier elimination procedures,
2. instantiation-based procedures for $\exists\forall$ formulas, and
3. synthesis procedures for single invocation conjectures without syntactic restrictions.

In particular, devising a sound and complete instantiation procedure for $\exists\forall$ formulas (Point 2) is sufficient both for quantifier elimination (Point 1) and for devising a sound and complete procedure for single invocation conjectures without syntactic restrictions (Point 3) [3]. Furthermore, a complete synthesis procedure for single invocation conjectures (Point 3) is trivially sufficient for devising a complete instantiation procedure (Point 2). Finally, many quantifier elimination techniques (Point 1), for instance based on virtual term substitution [22], can be rephrased as instantiation procedures (Point 2), although this direction of the correspondence does not necessarily hold in general. A number of previous works, e.g. [20, 34], are based on the correspondence between quantifier elimination and synthesis.

[3] For details, see [30].

4.1 Inferring When a Conjecture is Single Invocation

It is often the case that a conjecture is not single invocation but is equivalent to one that is. The latter can often be generated automatically from the former by a normalization process that includes normalizing the arguments of invocations across conjunctions, and using quantifier elimination to eliminate variables for which the function to synthesize is not applied.

Example 5. The (non-single invocation) synthesis conjecture $\exists f\, f(0) \approx 1 \wedge f(1) \approx 5$ is equivalent to the single invocation conjecture $\exists f \forall x\, (x \approx 0 \Rightarrow f(x) \approx 1) \wedge (x \approx 1 \Rightarrow f(x) \approx 5)$. □

Example 6. Consider the conjecture:

$$\exists f \forall x\, y\, z\, (x \geq y \wedge x \approx z) \vee (y \geq x \wedge y \approx z)) \Rightarrow f(x,y) \approx z$$

where f is of type $\mathsf{Int} \times \mathsf{Int} \rightarrow \mathsf{Int}$ and all other variables have type Int. Although f is only invoked once, this conjecture does not fit the single invocation pattern due to the additional quantification on z. However, quantification on z may be eliminated based on the following steps. First, replace the occurrence of $f(x,y)$ with a fresh variable w, and negate the conjecture to obtain:

$$(\forall w\ \exists x\, y)\exists z\ \neg((x \geq y \wedge x \approx z) \vee (y \geq x \wedge y \approx z)) \Rightarrow w \approx z)\ .$$

As this is a formula in linear arithmetic, we may use a quantifier elimination procedure to obtain an equivalent formula involving only w, x, and y such as:

$$\forall w\ \exists x y\, \neg((x \geq y \Rightarrow w \approx x) \wedge (y \geq x \Rightarrow w \approx y))\ .$$

From that, we can generate the following single invocation conjecture, which is provably equivalent to the original one:

$$\exists f \forall x\, y\, (x \geq y \Rightarrow f(x,y) \approx x) \wedge (y \geq x \Rightarrow f(x,y) \approx y)\ .$$

□

4.2 Counterexample-Guided Quantifier Instantiation for I/O Examples

While counterexample-guided quantifier instantiation is both highly efficient and complete for single invocation synthesis conjectures, it has the disadvantage of producing solutions that are suboptimal in terms of term size, especially in the case of partial specifications.

To see that, consider that based on Definition 1, all input-output example synthesis conjectures are also single invocation conjectures; thus, the techniques above are applicable, However, as demonstrated in the following example, synthesis by counterexample-guided quantifier instantiation produces for this class of conjectures sub-optimal solutions that, in a sense, overfit the specification.

Example 7. Consider the negated input-output example conjecture:

$$\neg\exists f \forall x\, (x \approx 1 \Rightarrow f(x) \approx 2) \wedge (x \approx 2 \Rightarrow f(x) \approx 3) \wedge (x \approx 7 \Rightarrow f(x) \approx 8)\ . \tag{5}$$

This formula is equivalent to the first-order formula:

$$\exists x \forall z\, \neg((x \approx 1 \Rightarrow z \approx 2) \wedge (x \approx 2 \Rightarrow z \approx 3) \wedge (x \approx 7 \Rightarrow z \approx 8))$$

which can be shown T-unsatisfiable with the following three instances:

$$\neg((x \approx 1 \Rightarrow 2 \approx 2) \wedge (x \approx 2 \Rightarrow 2 \approx 3) \wedge (x \approx 7 \Rightarrow 2 \approx 8)),$$
$$\neg((x \approx 1 \Rightarrow 3 \approx 2) \wedge (x \approx 2 \Rightarrow 3 \approx 3) \wedge (x \approx 7 \Rightarrow 3 \approx 8)),$$
$$\neg((x \approx 1 \Rightarrow 8 \approx 2) \wedge (x \approx 2 \Rightarrow 8 \approx 3) \wedge (x \approx 7 \Rightarrow 8 \approx 8))$$

which simplify to $x \approx 2 \vee x \approx 7$, $x \approx 1 \vee x \approx 7$ and $x \approx 1 \vee x \approx 2$, respectively. Hence:

$$\lambda x\, \mathsf{ite}\Big(\begin{array}{l}(x \approx 1 \Rightarrow 2 \approx 2) \wedge \\ (x \approx 2 \Rightarrow 2 \approx 3) \wedge \\ (x \approx 7 \Rightarrow 2 \approx 8)\end{array}, 2, \mathsf{ite}\Big(\begin{array}{l}(x \approx 1 \Rightarrow 3 \approx 2) \wedge \\ (x \approx 2 \Rightarrow 3 \approx 3) \wedge \\ (x \approx 7 \Rightarrow 3 \approx 8)\end{array}, 3, 8)\Big)$$

which simplifies to $\lambda x\, \mathsf{ite}(x \approx 1, 2, \mathsf{ite}(x \approx 2, 3, 8))$ is a solution for f in (16). In other words, counterexample-guided quantifier instantiation produces the trivial solution (consisting of an input-output table) for the given input-output example conjecture although shorter solutions, such as $\lambda x\, x+1$, exist. □

From this example, one can see that a more compelling use case of input-output example conjectures is when syntactic restrictions are imposed on the conjecture, which may force the synthesis procedure to produce more compact solutions. Indeed, the programming-by-examples paradigm [17] assumes such restrictions are provided so that intended solutions are generated by the underlying synthesis algorithm. There, the additional goal is also to capture function intended by the user by generalizing from a few input-output examples.

4.3 Counterexample-Guided Quantifier Instantiation with Syntactic Restrictions

To find solutions for single invocation synthesis conjectures in the presence of syntactic restrictions, a straightforward if incomplete technique is to first solve for the conjecture as before, ignoring the restrictions, and then check whether the generated solution satisfies the syntactic restrictions. If it does not, then we may try to reconstruct those portions of the solution that break the restrictions. We demonstrate this process in the following example.

Example 8. Consider again the synthesis conjecture from Example 4 but now assume we are additionally given syntactic restrictions expressed by this grammar with initial symbol I:

$$\begin{array}{lcl} \mathsf{I} & \to & 0 \mid 1 \mid x \mid y \mid \mathsf{I}+\mathsf{I} \mid \mathsf{ite}(\mathsf{B},\mathsf{I},\mathsf{I}) \\ \mathsf{B} & \to & \mathsf{I} > \mathsf{I} \mid \mathsf{I} \approx \mathsf{I} \mid \neg(\mathsf{B}) \end{array}$$

We may first ignore syntactic restrictions and run counterexample-guided quantifier instantiation on the conjecture, obtaining the solution $f = \lambda xy\, \mathsf{ite}(x \leq y+1, x+1, y+1)$ as before. This solution does not meet the syntactic restrictions above since the subterm $x \leq y+1$ cannot be generated from B. However, the term $\neg(x > y+1)$, which *is* generated from B, is equivalent to $x \leq y+1$ in T. It follows that $f = \lambda xy\, \mathsf{ite}(\neg(x > y+1), x+1, y+1)$ is also a solution for (2), which moreover satisfies the given syntactic restrictions. □

We refer to this technique as counterexample-guided quantifier instantiation with *solution reconstruction*.[4] Due to its heuristic nature, a synthesis procedure based on syntax-guided enumeration often has a higher success rate than one based on this approach. Thus, for single invocation properties with syntactic restrictions, we may use a portfolio approach that first tries counterexample-guided instantiation but aborts if solution reconstruction does not quickly succeed, and then resorts to syntax-guided enumeration, which we describe in detail in Section 5.

[4] For more details on this technique, see Section 5.2 of [29].

4.4 Synthesis via Quantifier Instantiation for Other Theories

While instantiation-based procedures for quantified linear arithmetic are now fairly mature [9, 12, 30], procedures for quantified constraints in other theories are still undergoing rapid development. Notably, current methods for quantified bit-vectors construct streams of instantiations based on candidate models [38], and use aggressive rewriting techniques to increase the likelihood that the problem can be solved during preprocessing. The limitation of current procedures is that they are often unable to construct useful symbolic instantiations required for finding concise proofs of unsatisfiability. To address this issue, a recent approach by Preiner et al. [26] uses syntax-guided synthesis to find relevant instantiations for quantified bit-vectors. An independent approach by Rabe et al. [27] uses new techniques to construct symbolic Skolem functions for quantified Boolean formulas.

Furthermore, a number of SMT solvers have been extended with theories outside the standard canon of traditional theories, such as unbounded strings and regular expressions [21], finite sets [5], and floating point arithmetic [10]. Devising counterexample-guided instantiation procedures for each of these theories remains an open challenge.

5 Synthesis via Syntax-Guided Enumeration

For some cases, counterexample-guided quantifier instantiation is either not applicable to the class of synthesis conjecture under consideration, or suboptimal because of user-provided syntactic restrictions. In these cases, we use enumerative syntax-guided techniques popularized by a number of recent synthesis tools, notably the enumerative solver by Udupa et al. [36, 4]. At a high level, these techniques consider a stream of candidate solutions sorted increasingly with respect to some total ordering, typically some linearization of term size. In this section, we focus on how these techniques can be performing using existing components of a DPLL(T)-based SMT solver. We will use the following running example.

Example 9. Consider the negated synthesis conjecture:

$$\neg\exists f\,\forall xy\,(f(x,y)\geq x\wedge f(x,y)\approx f(y,x)) \tag{6}$$

with syntactic restrictions on f provided by the following grammar with initial symbol I:

$$\begin{array}{lcl}\mathsf{I} & \rightarrow & 0 \mid x \mid y \mid \mathsf{I}+1 \mid \mathsf{ite}(\mathsf{B},\mathsf{I},\mathsf{I}) \\ \mathsf{B} & \rightarrow & \mathsf{I}\leq\mathsf{I} \mid \mathsf{I}\geq\mathsf{I} \mid \mathsf{I}\approx\mathsf{I} \mid \neg(\mathsf{B})\end{array}$$

Since SMT solvers do not natively have a notion of grammars, we rephrase the syntactic restrictions as a (set of) *algebraic datatypes*, for which a number of SMT solvers have dedicated decision procedures [7, 28]. Hence we construct the following set of (mutually recursive) algebraic datatypes:

$$\begin{array}{lcl}\mathsf{I} & = & 0 \mid \mathsf{x} \mid \mathsf{y} \mid \mathsf{plus}(\mathsf{I},\mathsf{I}_1) \mid \mathsf{if}(\mathsf{B},\mathsf{I},\mathsf{I}) \\ \mathsf{I}_1 & = & 1 \\ \mathsf{B} & = & \mathsf{leq}(\mathsf{I},\mathsf{I}) \mid \mathsf{eq}(\mathsf{I},\mathsf{I}) \mid \mathsf{not}(\mathsf{B})\end{array}$$

Here, the right hand side of each datatype lists the set of possible constructors for the datatype, where each constructor symbol corresponds to a symbol in the theory of arithmetic. Notice that this construction involved flattening, so that each construct consists of exactly one symbol applied to a set of arguments. Thus, the rule $\mathsf{I}\rightarrow\mathsf{I}+1$ required the auxiliary datatype I_1 with a single constructor 1. The construction also involved minimization, and hence the rule $\mathsf{B}\rightarrow\mathsf{I}\geq\mathsf{I}$ was discarded since it is redundant with respect

to $\mathsf{B} \to \mathsf{I} \leq \mathsf{I}$. For a datatype constant c, we call *analog of* c the term it corresponds to in the theory it encodes. For example, the analog of $\mathsf{plus}(\mathsf{x},1)$ is the arithmetic term $x+1$.

As described by Reynolds et al. [29], by using the datatypes $\mathsf{I}, \mathsf{I}_1, \mathsf{B}$, we can construct a first-order version of the formula (6) that encodes the syntactic restrictions on solutions for f. To do that, for τ in $(\mathsf{I}, \mathsf{I}_1, \mathsf{B})$, we first introduce an operator eval_τ or type $\tau \times \mathsf{Int} \times \mathsf{Int} \to \tau'$ where τ' is respectively in $(\mathsf{Int}, \mathsf{Int}, \mathsf{Bool})$. Informally, eval_τ is interpreted as a function that takes a datatype value d, an integer value a and an integer value b, and evaluates the analog of d under the assignment $\{x \mapsto a, y \mapsto b\}$. For example, $\mathsf{eval}_\mathsf{I}(\mathsf{x},2,3) = 2$; $\mathsf{eval}_\mathsf{I}(\mathsf{y},2,3) = 3$; $\mathsf{eval}_\mathsf{I}(\mathsf{plus}(\mathsf{x},1),2,3) = \mathsf{eval}_\mathsf{I}(\mathsf{x},2,3) + \mathsf{eval}_{\mathsf{I}_1}(1,2,3) = 2+1 = 3$; $\mathsf{eval}_\mathsf{I}(\mathsf{if}(\mathsf{leq}(\mathsf{x},\mathsf{y}),1,0),2,3) = \mathsf{ite}(\mathsf{eval}_\mathsf{B}(\mathsf{leq}(\mathsf{x},\mathsf{y}),2,3), \mathsf{eval}_\mathsf{I}(1,2,3), \mathsf{eval}_\mathsf{I}(0,2,3)) = 1$. Concretely, these semantics can be enforced in the SMT solver by implementing additional inference rules for unfolding and eliminating occurrences of the three eval_τ recursively over the datatypes I, I_1 and B.

Using these operators, we can construct a first-order formula corresponding to (6):

$$\neg \exists d_f \forall x\, y\, (\mathsf{eval}_\mathsf{I}(d_f,x,y) \geq x \wedge \mathsf{eval}_\mathsf{I}(d_f,x,y) \approx \mathsf{eval}_\mathsf{I}(d_f,y,x)) \tag{7}$$

where, however, d_f is a *first-order* variable of datatype sort I (corresponding to the second-order variable f of type $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$). This formula is equivalent to:

$$\forall d_f \exists x y \neg (\mathsf{eval}_\mathsf{I}(d_f,x,y) \geq x \wedge \mathsf{eval}_\mathsf{I}(d_f,x,y) \approx \mathsf{eval}_\mathsf{I}(d_f,y,x)) \tag{8}$$

which can be shown T-unsatisfiable by a single instantiation that maps d_f to $\mathsf{if}(\mathsf{leq}(\mathsf{y},\mathsf{x}),\mathsf{x},\mathsf{y})$. It is possible to prove that then $\lambda x y\, \mathsf{ite}(y \leq x, x, y)$ is a solution for f in (6). □

Like the previous section, the key technical challenge in the above technique is determining the datatype term $\mathsf{if}(\mathsf{leq}(\mathsf{y},\mathsf{x}),\mathsf{x},\mathsf{y})$ used to instantiate the universal quantifier in (8). At a high level, this is done brute-force, by considering a stream of candidate terms based on some ordering.[5] Such candidates can be generated by an extension of the decision procedure for quantifier-free equational constraints over algebraic datatypes which is implementend in some SMT solvers [7, 28]. Before giving more detail on the candidate generation procedure, we must introduce some more notation.

If d is a term of some datatype type D and C is a constructor of D, we write $\mathsf{is}_\mathsf{C}(d)$ to denote a predicate that is satisfied exactly when d is interpreted as a datatype value whose top symbol is C. These predicates are sometimes referred to as *discriminators*. We write $\mathsf{sel}_{\tau,n}(d)$ to denote the term that is interpreted as the n^{th} child of d that has type τ if one exists, or is freely interpreted otherwise. In Example 9, if $d^\mathscr{I} = \mathsf{plus}(\mathsf{x},1)$, then $\mathsf{sel}_{\mathsf{I},1}(d)^\mathscr{I} = \mathsf{x}$ and $\mathsf{sel}_{\mathsf{I}_1,1}(d)^\mathscr{I} = 1$, whereas if $d^\mathscr{I} = \mathsf{if}(\mathsf{leq}(\mathsf{x},\mathsf{y}),\mathsf{y},\mathsf{x})$, then $\mathsf{sel}_{\mathsf{B},1}(d)^\mathscr{I} = \mathsf{leq}(\mathsf{x},\mathsf{y})$, $\mathsf{sel}_{\mathsf{I},2}(d)^\mathscr{I} = \mathsf{sel}_{\mathsf{I},1}(\mathsf{sel}_{\mathsf{B},2}(d))^\mathscr{I} = \mathsf{x}$ and $\mathsf{sel}_{\mathsf{I},1}(d)^\mathscr{I} = \mathsf{sel}_{\mathsf{I},2}(\mathsf{sel}_{\mathsf{B},1}(d))^\mathscr{I} = \mathsf{y}$. The functions $\mathsf{sel}_{\tau,n}$ are often referred to as *selectors*, where here we allow that selector symbols may access the subfields of multiple constructors[6].

Now, consider again the algebraic datatypes $\mathsf{I}, \mathsf{I}_1, \mathsf{B}$ from Example 9, and let d be a fresh variable of type I. A DPLL(T)-based SMT solver may be configured to enumerate a stream of values of datatype I by finding models $\mathscr{I}$ for a evolving set of clauses Γ_d, initially empty. On each iteration, assuming Γ_d is satisfied by some interpretation $\mathscr{I}$, we consider the value of $d^\mathscr{I}$ as an instantiation for d_f in (8). The value of $d^\mathscr{I}$ either corresponds to a solution for f in (6), in which case the resulting instantiation simplifies to false, or it does not, in which case the resulting instantiation simplifies to true. In the latter case, we ensure all subsequent models $\mathscr{J}$ for d are such that $d^\mathscr{J} \neq d^\mathscr{I}$. To do so, we add a clause of the form $\neg \mathsf{is}_{\mathsf{C}_1}(t_1) \vee \ldots \vee \neg \mathsf{is}_{\mathsf{C}_n}(t_n)$ where each t_i is a (possibly empty) chain of selectors applied to d.

[5] This ordering is determined, for instance, by counting the number of non-nullary constructors in the terms.

[6] In most presentation of algebraic datatypes [7, 28], selectors are associated with one constructor only. Decision procedures for datatypes can be easily modified to support our version of selectors and discriminators.

For example, if we find that the instantiation of (8) that maps d_f to $d^{\mathscr{I}} = \mathsf{plus}(\mathsf{x},1)$ is not a solution, then we add the clause:

$$\neg \mathsf{is}_{\mathsf{plus}}(d) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(d)) \vee \neg \mathsf{is}_{\mathsf{1}}(\mathsf{sel}_{\mathsf{I}_1,2}(d)) \tag{9}$$

to Γ_d. This clause records the fact that we want subsequent models $\mathscr{J}$ of Γ_d to be such that $d^{\mathscr{J}} \neq \mathsf{plus}(\mathsf{x},1)$, since at least one of the disjuncts in the above formula must hold.

5.1 Symmetry Breaking Based on Theory-Specific Simplification

The key optimization for making enumerative syntax-guided search scalable in practice is to avoid considering multiple solutions that are identifiable based on a suitable equivalence relation. For this purpose, we leverage the fact that DPLL(T)-based SMT solvers use simplification techniques for terms and formulas. Such techniques simplify the development of decision procedures and can be used to improve the performance of subsolvers for certain background theories [32]. The same simplification techniques can be used in turn to recognize when a candidate solution to equivalent to another, thereby allowing us to prune portions of the enumerative search.

Specifically, state-of-the-art SMT solvers are capable of constructing from any term t a *simplified form*[7] that we denote here as $t{\downarrow}$. This is a term that is equivalent to t in the background theory T, but is simpler by some measure. We do not need to require simplified forms to be unique. Specifically, equivalent terms s and t may have different simplified forms $s{\downarrow}$ and $t{\downarrow}$. However, the more equivalent terms that have the same simplified form, the more effective our pruning technique is. We illustrate how theory-specific simplification methods can be used to prune search space in the following example.

Example 10. Consider the algebraic datatypes:

$$\begin{array}{lcl} \mathsf{I} & = & \mathsf{0} \mid \mathsf{1} \mid \mathsf{x} \mid \mathsf{y} \mid \mathsf{plus}(\mathsf{I},\mathsf{I}) \mid \mathsf{if}(\mathsf{B},\mathsf{I},\mathsf{I}) \\ \mathsf{B} & = & \mathsf{leq}(\mathsf{I},\mathsf{I}) \mid \mathsf{eq}(\mathsf{I},\mathsf{I}) \mid \mathsf{not}(\mathsf{B}) \end{array}$$

Given an interpretation $\mathscr{I}$, consider a list of candidate solutions $d^{\mathscr{I}}$, their arithmetic analog, and their corresponding simplified form obtained with simplifications specific to integer arithmetic:

#	$d^{\mathscr{I}}$	Analog	Analog$\downarrow$
1	x	x	x
	...		
2	$\mathsf{plus}(\mathsf{y},1)$	$y+1$	$1+y$
3	$\mathsf{plus}(\mathsf{x},0)$	$x+0$	x
4	$\mathsf{plus}(1,\mathsf{y})$	$1+y$	$1+y$
	...		
5	$\mathsf{if}(\mathsf{leq}(0,1),\mathsf{x},0)$	$\mathsf{ite}(0 \leq 1, x, 0)$	x

In this table, we assume our simplification orders monomial sums based on some ordering so that, for instance, $(y+1){\downarrow} = 1+y$. Notice that the third, fourth and fifth terms listed in this enumeration have analogs that are identical after simplification to previous terms in the enumeration. For example, the simplified analog of $\mathsf{plus}(\mathsf{x},0)$ is the same as x. For this reason, we may add clauses to the SMT solver

[7] This is sometimes also called its normal or rewritten form.

that exclude models $\mathscr{I}$ where d is interpreted as $\mathsf{plus}(\mathsf{x},0)$. In particular, the clauses:

$$\neg \mathsf{is}_{\mathsf{plus}}(d) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(d)) \vee \neg \mathsf{is}_{\mathsf{0}}(\mathsf{sel}_{\mathsf{I},2}(d)) \tag{10}$$

$$\neg \mathsf{is}_{\mathsf{plus}}(d) \vee \neg \mathsf{is}_{\mathsf{1}}(\mathsf{sel}_{\mathsf{I},1}(d)) \vee \neg \mathsf{is}_{\mathsf{y}}(\mathsf{sel}_{\mathsf{I},2}(d)) \tag{11}$$

$$\begin{aligned}\neg \mathsf{is}_{\mathsf{if}}(d) \vee \neg \mathsf{is}_{\mathsf{leq}}(\mathsf{sel}_{\mathsf{B},1}(d)) \vee \neg \mathsf{is}_{\mathsf{0}}(\mathsf{sel}_{\mathsf{I},1}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \neg \mathsf{is}_{\mathsf{1}}(\mathsf{sel}_{\mathsf{I},2}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \\ \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(d)) \vee \neg \mathsf{is}_{\mathsf{0}}(\mathsf{sel}_{\mathsf{I},2}(d))\end{aligned} \tag{12}$$

exclude models where d is interpreted as $\mathsf{plus}(\mathsf{x},0)$, $\mathsf{plus}(1,\mathsf{y})$, and $\mathsf{if}(\mathsf{leq}(0,1),\mathsf{x},\mathsf{y})$ respectively. We refer to formulas (10-12) as *symmetry breaking clauses*. Notice that symmetry breaking clauses do not rule out interesting candidate solutions since they only have the effect of avoiding repeated solutions modulo equivalence in T. □

We extend the quantifier-free decision procedure for algebraic datatypes to maintain a database of entries of form described in the table in Example 10. Symmetry breaking clauses are added to the set Γ_d (introduced earlier) in two ways. First, whenever a new datatype term t is generated, we eagerly generate a predetermined set of symmetry breaking clauses to restrict the value of t in all models, where this set is determined by statically analyzing the set of constructors and the symbols in the theory they correspond to. Second, when considering an interpretation $\mathscr{I}$, if any datatype term $t^{\mathscr{I}}$ has a value that is not unique with respect to existing values in our database, we add a symmetry breaking clause that implies that $t \not\approx t^{\mathscr{I}}$, thereby forcing the solver to find a new interpretation.

Notice that symmetry breaking clauses can be reused for multiple terms of the same type. For example, our implementation may consider a modified version of the symmetry breaking clause (10) where d is replaced by $\mathsf{sel}_{\mathsf{I},1}(d)$:

$$\neg \mathsf{is}_{\mathsf{plus}}(\mathsf{sel}_{\mathsf{I},1}(d)) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(\mathsf{sel}_{\mathsf{I},1}(d))) \vee \neg \mathsf{is}_{\mathsf{0}}(\mathsf{sel}_{\mathsf{I},2}(\mathsf{sel}_{\mathsf{I},1}(d))) \tag{13}$$

This clause can be added to Γ_d, thereby enabling the solver to avoid candidate models where the first child of d with type I is interpreted as $\mathsf{plus}(\mathsf{x},0)$. This further enables the solver to avoid candidate solutions like $\mathsf{plus}(\mathsf{plus}(\mathsf{x},0),\mathsf{y})$. A similar observation is made in the approach by Alur et al. [4].

Symmetry breaking clauses may be generalized in several ways to consider, for instance, cases when a term simplifies to one of its proper subterms, or when simplification can be performed independently of certain subterms.

Example 11. Recall that the symmetry breaking clause (10) in Example 10 was justified by the fact that $(x+0)\downarrow = x$, which has the same simplified form as the analog of x. Since $(z+0)\downarrow = z$ for fresh variable z, we may conclude that $(t+0)\downarrow = t$ for all terms t. Thus, we may generalize this symmetry breaking clause to exclude the disjunct that depends on x, obtaining the clause:

$$\neg \mathsf{is}_{\mathsf{plus}}(d) \vee \neg \mathsf{is}_{\mathsf{0}}(\mathsf{sel}_{\mathsf{I},2}(d)) \tag{14}$$

This clause rules out all models where d is interpreted as a value of the form $\mathsf{plus}(d_0,0)$ *for any* d_0. It can be shown that this clause does not rule out any interesting candidate solutions, and hence can be used to avoid an (infinite) class of candidate solutions. □

Example 12. The symmetry breaking clause (12) in Example 10 was justified by the fact that $\mathsf{ite}(0 \leq 1,x,0)\downarrow = x$. We can generalize this observation to terms of the form $\mathsf{ite}(0 \leq 1,x,t)$ by noting that $\mathsf{ite}(0 \leq 1,x,z)\downarrow = x$ for fresh variable z. Thus, the simplification in this example did not depend on the second occurrence of 0. Hence, we can generalize (12) to exclude its corresponding disjunct $\neg \mathsf{is}_{\mathsf{0}}(\mathsf{sel}_{\mathsf{I},2}(d))$, obtaining the clause:

$$\neg \mathsf{is}_{\mathsf{if}}(d) \vee \neg \mathsf{is}_{\mathsf{leq}}(\mathsf{sel}_{\mathsf{B},1}(d)) \vee \neg \mathsf{is}_{\mathsf{0}}(\mathsf{sel}_{\mathsf{I},1}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \neg \mathsf{is}_{\mathsf{1}}(\mathsf{sel}_{\mathsf{I},2}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(d)) \tag{15}$$

This clause rules out all models where d is interpreted as a value of the form $\mathsf{if}(\mathsf{leq}(0,1),\mathsf{x},d_0)$ for any d_0. We can generalize this further by noting $\mathsf{ite}(0 \leq 1, z_1, z_2)\downarrow = z_1$ for fresh variables z_1, z_2. Hence we may drop the disjunct $\neg\mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(d))$ from (15) as well. □

Generalizing symmetry breaking clauses has a dramatic positive impact on the performance of syntax-guided enumerative search in our implementation. Notice that the above generalization techniques must take syntactic restrictions into account. Consider the algebraic datatypes:

$$\begin{array}{lcl} \mathsf{I} & = & \mathsf{x} \mid \mathsf{plus}(\mathsf{I}_1, \mathsf{I}_2) \\ \mathsf{I}_1 & = & 0 \mid 1 \mid \mathsf{x} \mid \mathsf{y} \mid \mathsf{plus}(\mathsf{I},\mathsf{I}) \\ \mathsf{I}_2 & = & 0 \end{array}$$

Although the terms x and $\mathsf{plus}(\mathsf{x},0)$ have identical analogs after simplification, the symmetry breaking clause for $d : \mathsf{I}$ of the form $\neg\mathsf{is}_{\mathsf{plus}}(d) \vee \neg\mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I}_1,1}(d)) \vee \neg\mathsf{is}_0(\mathsf{sel}_{\mathsf{I}_2,1}(d))$ cannot be generalized to $\neg\mathsf{is}_{\mathsf{plus}}(d) \vee \neg\mathsf{is}_0(\mathsf{sel}_{\mathsf{I}_2,1}(d))$. Even though $(z+0)\downarrow = z$ for fresh variable z, this clause clearly rules out possible solutions for d like $\mathsf{plus}(\mathsf{y},0)$ that could not be constructed if such a symmetry breaking clause were used. In practice, we annotate terms with the datatype they correspond to. In this example, $z+0$ is annotated with I and z is annotated with I_1. Hence, we can infer that the simplification $(z+0)\downarrow = z$ does not preserve syntactic restrictions since the annotated types of $z+0$ and z are not identical.

5.2 Symmetry Breaking Based on I/O Example Evaluation

Alur et al. [4] note that enumerative syntax-guided search need only consider candidate solutions that are unique when evaluated on the set of input points in the conjecture. Thus, we may use an even stronger criterion for recognizing when candidate solutions can be discarded. We demonstrate how this observation can be incorporated in our setting in the following example.

Example 13. Consider the input-output example conjecture:

$$\begin{array}{l} \neg\exists f \forall x y \ (x \approx 1 \wedge y \approx 0 \Rightarrow f(x,y) \approx 1) \wedge \\ \qquad\qquad (x \approx 2 \wedge y \approx 1 \Rightarrow f(x,y) \approx 3) \wedge \\ \qquad\qquad (x \approx 7 \wedge y \approx 1 \Rightarrow f(x,y) \approx 8) \end{array} \tag{16}$$

where f is of type $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ and all other variables are of type Int. Note that this conjecture consists of three conjunctions which correspond to the input points $(x,y) = (1,1)$, $(2,1)$, and $(7,1)$. Assume we are given syntactic restrictions for this conjecture, which we transform into the algebraic datatype:

$$\begin{array}{lcl} \mathsf{I} & = & 0 \mid 1 \mid \mathsf{x} \mid \mathsf{y} \mid \mathsf{plus}(\mathsf{I},\mathsf{I}) \mid \mathsf{if}(\mathsf{B},\mathsf{I},\mathsf{I}) \\ \mathsf{B} & = & \mathsf{leq}(\mathsf{I},\mathsf{I}) \mid \mathsf{eq}(\mathsf{I},\mathsf{I}) \mid \mathsf{not}(\mathsf{B}) \end{array}$$

For each candidate solution $d^{\mathscr{I}}$, we again compute its arithmetic analog, and its analog after theory-specific simplification; but now also compute its evaluation on the three input points of the conjecture:

#	$d^{\mathscr{I}}$	Analog	Analog↓	Eval on Examples
1	1	1	1	1,1,1
2	x	x	x	1,2,7
	...			
3	plus(x,x)	$x+x$	$1+y$	2,4,14
4	plus(x,0)	$x+0$	x	n/a
	...			
5	if(leq(y,x),x,y)	ite($y \leq x, x, y$)	ite($y \leq x, x, y$)	1,2,7
6	if(leq(1,y),1,x)	ite($1 \leq y, 1, x$)	ite($1 \leq y, 1, x$)	1,1,1

We are interested in avoiding solutions that are either not unique after simplification, or not unique when evaluated on input examples. Consider the fifth term $\mathsf{if}(\mathsf{leq}(\mathsf{y},\mathsf{x}),\mathsf{x},\mathsf{y})$, whose analog after simplification is $\mathsf{ite}(y \leq x, x, y)$, which is unique with respect to the previous terms listed in the enumeration. However, evaluating $\mathsf{ite}(y \leq x, x, y)$ on the points $(x,y) = (1,1)$, $(2,1)$, and $(7,1)$ gives 1, 2, and 7 respectively. Thus, with respect to the conjecture (16), the candidate solutions x and $\mathsf{if}(\mathsf{leq}(\mathsf{y},\mathsf{x}),\mathsf{x},\mathsf{y})$ are identical, and hence we may discard the latter using a symmetry breaking clause:

$$\neg \mathsf{is}_{\mathsf{if}}(d) \vee \neg \mathsf{is}_{\mathsf{leq}}(\mathsf{sel}_{\mathsf{B},1}(d)) \vee \neg \mathsf{is}_{\mathsf{y}}(\mathsf{sel}_{\mathsf{I},1}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},2}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(d)) \vee \neg \mathsf{is}_{\mathsf{y}}(\mathsf{sel}_{\mathsf{I},2}(d)) \tag{17}$$

The candidate solutions 1 and $\mathsf{if}(\mathsf{leq}(1,\mathsf{y}),1,\mathsf{x})$ are identical for similar reasons. Observe that for the term $\mathsf{plus}(\mathsf{x},0)$, we find that its analog after simplification is equivalent to x, and hence we do not need to compute its evaluation on the input examples since they are guaranteed to be identical. □

We may use symmetry breaking clauses to eliminate candidate solutions that are not unique when considering input examples. We may use the same enhancements from the previous section for these clauses as well, namely, they can be reapplied to any term of the proper type and generalized using similar techniques. For example, the clause (17) can be generalized to:

$$\neg \mathsf{is}_{\mathsf{if}}(d) \vee \neg \mathsf{is}_{\mathsf{leq}}(\mathsf{sel}_{\mathsf{B},1}(d)) \vee \neg \mathsf{is}_{\mathsf{y}}(\mathsf{sel}_{\mathsf{I},1}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},2}(\mathsf{sel}_{\mathsf{B},1}(d))) \vee \neg \mathsf{is}_{\mathsf{x}}(\mathsf{sel}_{\mathsf{I},1}(d)) \tag{18}$$

by noting that evaluating the term $\mathsf{ite}(y \leq x, x, z)$ on the points $(x,y) = (1,1)$, $(2,1)$, and $(7,1)$ gives $1,2,7$ respectively for fresh variable z. Hence we may drop the disjunct $\neg \mathsf{is}_{\mathsf{y}}(\mathsf{sel}_{\mathsf{I},2}(d))$ since this reasoning did not depend on the second occurrence of y in the candidate solution $\mathsf{if}(\mathsf{leq}(\mathsf{y},\mathsf{x}),\mathsf{x},\mathsf{y})$.

6 Conclusions and Future Work

We have presented techniques for integrating techniques for syntax-guided synthesis in the core of a DPLL(T)-based SMT solver. All techniques described in the paper are implemented in the open-source SMT solver CVC4. For future work, we are investigating cases where synthesis by quantifier instantiation and synthesis by syntax-guided enumeration can be combined, as well as investigating efficient quantifier instantiation techniques for new background theories.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2013): *Syntax-guided synthesis*. In: *FMCAD*, IEEE, pp. 1–17.

[2] Rajeev Alur, Dana Fisman, Rishabh Singh & Armando Solar-Lezama (2016): *Results and Analysis of SyGuS-Comp'15. arXiv preprint arXiv:1602.01170.*

[3] Rajeev Alur, Dana Fisman, Rishabh Singh & Armando Solar-Lezama (2016): *SyGuS-Comp 2016: Results and Analysis*. In: *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, pp. 178–202.

[4] Rajeev Alur, Arjun Radhakrishna & Abhishek Udupa (2017): *Scaling Enumerative Program Synthesis via Divide and Conquer*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 319–336.

[5] Kshitij Bansal, Andrew Reynolds, Clark Barrett & Cesare Tinelli (2016): *A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT*. In Nicola Olivetti & Ashish Tiwari, editors: *Proceedings of the 8th International Joint Conference on Automated Reasoning*, 9706, pp. 82–98.

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[7] Clark Barrett, Igor Shikanian & Cesare Tinelli (2007): *An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types*. *Electr. Notes Theor. Comput. Sci.* 174(8), pp. 23–37.

[8] Nikolaj Bjørner (2010): *Linear Quantifier Elimination as an Abstract Decision Procedure*. In Jürgen Giesl & Reiner Hähnle, editors: *IJCAR*, *LNCS* 6173, Springer, pp. 316–330, doi:10.1007/978-3-642-14203-1_27.

[9] Nikolaj Bjørner & Mikolás Janota (2015): *Playing with Quantified Satisfaction*. In: *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015.*, pp. 15–27.

[10] Martin Brain, Vijay DSilva, Alberto Griggio, Leopold Haller & Daniel Kroening (2014): *Deciding Floating-point Logic with Abstract Conflict Driven Clause Learning*. *Formal Methods in System Design* 45(2), pp. 213–245.

[11] D. C. Cooper (1972): *Theorem Proving in Arithmetic without Multiplication*. In: *Machine Intelligence, pages 91100.*

[12] Bruno Dutertre (2015): *Solving Exists/Forall Problems With Yices*. In: *Workshop on Satisfiability Modulo Theories.*

[13] Azadeh Farzan & Zachary Kincaid (2016): *Linear Arithmetic Satisfiability via Strategy Improvement*. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pp. 735–743.

[14] Grigory Fedyukovich, Arie Gurfinkel & Natasha Sharygina (2015): *Automated Discovery of Simulation Between Programs*. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pp. 606–621.

[15] Jeanne Ferrante & Charles W. Rackoff (1979): *The Computational Complexity of Logical Theories*. *Lecture Notes in Mathematics* 718, Springer.

[16] Harald Ganzinger & Konstantin Korovin (2003): *New directions in instantiation-based theorem proving*. In: *Logic in Computer Science, 2003.*, IEEE.

[17] Sumit Gulwani (2011): *Automating string processing in spreadsheets using input-output examples*. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pp. 317–330.

[18] Sumit Gulwani, Susmit Jha, Ashish Tiwari & Ramarathnam Venkatesan (2011): *Synthesis of loop-free programs*. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pp. 62–73, doi:10.1145/1993498.1993506.

[19] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2014): *SMT-based Model Checking for Recursive Programs*. In: *Computer Aided Verification*, Springer International Publishing.

[20] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac & Philippe Suter (2010): *Complete functional synthesis*. In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pp. 316–329.

[21] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett & Morgan Deters (2014): *A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions*. In: *Computer Aided Verification - 26th International Conference, CAV 2014*, pp. 646–662, doi:10.1007/978-3-319-08867-9_43.

[22] Rüdiger Loos & Volker Weispfenning (1993): *Applying Linear Quantifier Elimination*. *Comput. J.* 36(5), pp. 450–462, doi:10.1093/comjnl/36.5.450.

[23] David Monniaux (2010): *Quantifier Elimination by Lazy Model Enumeration*. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pp. 585–599, doi:10.1007/978-3-642-14295-6_51.

[24] Leonardo Mendonça de Moura & Nikolaj Bjørner (2007): *Efficient E-Matching for SMT Solvers*. In Frank Pfenning, editor: *CADE*, *LNCS* 4603, Springer, pp. 183–198, doi:10.1007/978-3-540-73595-3_13.

[25] Robert Nieuwenhuis, Albert Oliveras & Cesare Tinelli (2006): *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*)*. *J. ACM* 53(6), pp. 937–977, doi:10.1145/1217856.1217859.

[26] Mathias Preiner, Aina Niemetz & Armin Biere (2017): *Counterexample-Guided Model Synthesis*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 264–280.

[27] Markus N. Rabe & Sanjit A. Seshia (2016): *Incremental Determinization*. In: *Theory and Applications of Satisfiability Testing - SAT*, pp. 375–392.

[28] Andrew Reynolds & Jasmin Christian Blanchette (2015): *A Decision Procedure for (co) datatypes in SMT Solvers*. In: *International Conference on Automated Deduction*, Springer International Publishing, pp. 197–213.

[29] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli & Clark W. Barrett (2015): *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*, pp. 198–216, doi:10.1007/978-3-319-21668-3_12.

[30] Andrew Reynolds, Tim King & Viktor Kuncak (2017): *Solving quantified linear arithmetic by counterexample-guided instantiation*. *Formal Methods in System Design*.

[31] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett & Morgan Deters (2017): *Refutation-based synthesis in SMT*. *Formal Methods in System Design*.

[32] Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang & Cesare Tinelli (2017): *Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification*. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pp. 453–474.

[33] Armando Solar-Lezama (2013): *Program sketching*. *STTT* 15(5-6), pp. 475–495, doi:10.1007/s10009-012-0249-7.

[34] Thomas Sturm & Ashish Tiwari (2011): *Verification and Synthesis using Real Quantifier Elimination*. In: *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, ACM, pp. 329–336.

[35] Ashish Tiwari, Adria Gascón & Bruno Dutertre (2015): *Program Synthesis using Dual Interpretation*. In: *International Conference on Automated Deduction*, Springer International Publishing, pp. 482–497.

[36] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin & Rajeev Alur (2013): *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*, pp. 287–296, doi:10.1145/2462156.2462174.

[37] Yue Wang, Neil T. Dantam, Swarat Chaudhuri & Lydia E. Kavraki (2016): *Task and Motion Policy Synthesis as Liveness Games*. In: *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, p. 536.

[38] Christoph M Wintersteiger, Youssef Hamadi & Leonardo De Moura (2013): *Efficiently Solving Quantified Bit-vector Formulas*. *Formal Methods in System Design* 42(1), pp. 3–23.