Counterexample Guided Quantifier Instantiation for Synthesis in SMT

Andrew Reynolds[1], Morgan Deters[2],
Viktor Kuncak[1], Clark Barrett[2], and Cesare Tinelli[3]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2] Department of Computer Science, New York University
[3] Department of Computer Science, The University of Iowa

Abstract. This paper presents the first program synthesis engine implemented inside an SMT solver. We formulate our technique as support for *synthesis conjectures*. We present an approach that extracts solution functions from unsatisfiability proofs of the negated form of synthesis conjectures. To make finding such proofs feasible, we present counterexample-guided techniques for quantifier instantiation. A particularly important class of specifications are single-invocation properties, for which we present a dedicated algorithm. To support syntax restrictions on generated solutions, our approach can transform a solution found without restrictions into the desired syntactic format. As an alternative, we show how to use evaluation function axioms to embed syntactic restrictions into constraints over inductive datatypes, then use an inductive data type decision procedure to drive synthesis. We describe several techniques for pruning the search space of solutions, which improve scalability. Our experimental evaluation on syntax-guided synthesis benchmarks shows that our implementation in CVC4 is competitive with state-of-the-art tools for synthesis.

1 Introduction

Synthesis of functions that meet a given specification is a long-standing fundamental objective that has received great attention recently. This functionality directly applies to synthesis of functional programs [19, 20] and also translates to imperative programs through techniques that include bounding input space, verification condition generation, and invariant discovery [30, 31, 33]. Moreover, this task is an important subroutine in synthesis of protocols and reactive systems, especially when these systems are infinite-state [3, 29]. The SyGuS format and competition [1, 2, 25] inspired by the success of the SMT-LIB and SMT-COMP efforts [5], has significantly improved and simplified the process of rigorously comparing different solvers on synthesis problems.

Connection between synthesis and theorem proving was established already in early work on the subject [14, 22]. It is notable that early research [22] found that the capabilities of theorem provers were the main bottleneck for synthesis. Taking lessons from automated software verification, recent work on synthesis has made use of advances in theorem proving, particularly in SAT and SMT solvers. However, that work avoids formulating the overall synthesis task as a theorem proving problem directly. Instead, existing work typically builds custom loops outside of an SMT solver or a SAT solver,

often using numerous variants of counterexample-guided synthesis. A typical role of the SMT solver has been to validate candidate solutions and provide counterexamples that guide subsequent search, although approaches such as symbolic term exploration [17] also use an SMT solver to explore a representation of the space of solutions. In existing approaches, SMT solvers thus receive a large number of separate queries, with limited communication between these different steps.

Contributions. In this paper, we revisit the formulation of the overall synthesis task as a theorem proving problem. We observe that SMT solvers already have some of the key functionality for synthesis; we show how to improve the existing and introduce new algorithms to make SMT-based synthesis competitive with the state of the art. The resulting implementation outperforms state of the art substantially on important classes of synthesis problems. Our specific contributions are the following.

- We show how to formulate an important class of synthesis problems as disproving universally quantified formulas, and how to automatically extract synthesized functions from quantifier instantiations.
- We present counterexample-guided techniques for quantifier instantiation, which are crucial to obtain competitive performance on synthesis tasks.
- We present several techniques to simplify functions extracted from quantifier instantiations, helping ensure that synthesized functions are small and that they adhere to specified syntactic requirements.
- We show how to encode syntactic restrictions using theories of inductive data types and axiomatizable evaluation functions. This results in a flexible procedure that has desirable theoretical properties and also shows promise in practice.
- We show that for an important class of single-invocation properties (that is, synthesis of functions from relations), the implementation of our approach in CVC4 significantly outperforms leading tools from the SyGuS competition.

2 Synthesis inside an SMT Solver

We are interested in synthesizing computable functions automatically from formal logical specifications stating properties of these functions. Since synthesis involves finding (and so proving the existence) of functions, we use notions from many-sorted second-order logic to define the general problem. In this section we present several techniques that allow us to formulate a version of the synthesis problem in *first-order logic* alone, which is the starting point for allowing us to tackle the problem using SMT solvers.

2.1 Preliminaries

We fix a set $\mathbf{S}$ of *sort symbols* and an (infix) equality predicate $\approx$ of type $\sigma \times \sigma$ for each $\sigma \in \mathbf{S}$, which we always interpret as the identity relation over (the set denoted by) σ. For every non-empty sort sequence $\boldsymbol{\sigma} \in \mathbf{S}^+$ with $\boldsymbol{\sigma} = \sigma_1 \cdots \sigma_n \sigma$, we fix an infinite set $\mathbf{X}_{\boldsymbol{\sigma}}$ of *variables* $x^{\sigma_1 \cdots \sigma_n \sigma}$ *of type* $\sigma_1 \times \cdots \times \sigma_n \to \sigma$. For each sort σ we identity the type $() \to \sigma$ with σ and call it a *first-order type*. We assume the sets $\mathbf{X}_{\boldsymbol{\sigma}}$ are pairwise disjoint and let $\mathbf{X}$ be their union. A *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq \mathbf{S}$

of sort symbols and a set Σ^{f} of *function symbols* $f^{\sigma_1 \cdots \sigma_n \sigma}$ *of type* $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, where $n \geq 0$ and $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma^{\mathrm{s}}$. When n above is 0, we call f a *constant symbol*. We drop the sort superscript from variables or function symbols when it is clear from context or unimportant. We assume that signatures always include a Boolean sort Bool and constants $\top$ and $\bot$ of type Bool (respectively, for true and false). Given a many-sorted signature Σ together with quantifiers and lambda abstraction, the notion of well-sorted term, atom, literal, clause, and formula with variables in $\mathbf{X}$ are defined as usual in second-order logic; they are referred to respectively as *Σ-terms*, *Σ-atoms* and so on.[4] A Σ-term/formula is *ground* if it has no variables, it is *first-order* if it has only *first-order variables*, that is, variables of first-order type. When $\boldsymbol{x} = (x_1, \ldots, x_n)$ is a tuple of variables and Q is either $\forall$ or $\exists$, we write $Q\boldsymbol{x}\, \varphi$ as an abbreviation of $Qx_1 \cdots Qx_n\, \varphi$. If e is a Σ-term or formula and $\boldsymbol{x}$ has no repeated variables, we write $e[\boldsymbol{x}]$ to denote that e's free variables are from $\boldsymbol{x}$; if $\boldsymbol{s} = (s_1, \ldots, s_n)$ and $\boldsymbol{t} = (t_1, \ldots, t_n)$ are term tuples, we write $e[\boldsymbol{t}]$ for the term or formula obtained from e by simultaneously replacing each occurrence of x_i in e by t_i. A *Σ-interpretation* $\mathcal{I}$ maps: each $\sigma \in \Sigma^{\mathrm{s}}$ to a non-empty set $\sigma^{\mathcal{I}}$, the *domain* of σ in $\mathcal{I}$, with $\mathsf{Bool}^{\mathcal{I}} = \{\top, \bot\}$; each $u^{\sigma_1 \cdots \sigma_n \sigma}$ in $\mathbf{X}$ or in Σ^{f} to a total function $u^{\mathcal{I}} : \sigma_1^{\mathcal{I}} \times \cdots \times \sigma_n^{\mathcal{I}} \to \sigma^{\mathcal{I}}$ when $n > 0$ and to an element of $\sigma^{\mathcal{I}}$ when $n = 0$. If $x_1, \ldots, x_n$ are variables and $v_1, \ldots, v_n$ are well-typed values for them, we denote by $\mathcal{I}[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ the Σ-interpretation that maps each x_i to v_i and is otherwise identical to $\mathcal{I}$. A satisfiability relation between Σ-interpretations and Σ-formulas is defined inductively as usual. A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a non-empty class of Σ-interpretations, the *models* of T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only in how it interprets the variables is also in $\mathbf{I}$) and isomorphism. A Σ-formula $\varphi[\boldsymbol{x}]$ is *T-satisfiable* (resp., *T-unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A satisfying interpretation for φ *models (or is a model of)* φ. A formula φ is *T-valid*, written $\models_T \varphi$, if every model of T is a model of φ. Given a class (i.e., a set in our case) $\mathbf{L}$ of Σ-formulas, a Σ-theory T is *satisfaction complete* wrt to $\mathbf{L}$ if every T-satisfiable formula of $\mathbf{L}$ is T-valid. In this paper we will consider only theories that are satisfaction complete wrt the formulas we are interested in. Most theories used in SMT (in particular, all theories of a specific structure such various theories of the integers, the reals, the strings, the inductive datatypes, the bit vectors, and so on) are satisfaction complete with respect to the class of closed first-order Σ-formulas. Other theories, such as the theory of arrays are satisfaction complete only wrt to severely restricted classes of formulas (the benchmarks we had available for evaluation did not contain arrays).

2.2 Synthesis Conjectures

We consider the synthesis problem in the context of some theory T of signature Σ that allows us to provide the function's specification as a Σ-formula. Specifically, we will consider *synthesis conjectures* expressed as (well-sorted) formulas of the form

$$\exists f^{\sigma_1 \cdots \sigma_n \sigma}\, \forall x_1^{\sigma_1}\, \cdots\, \forall x_n^{\sigma_n}\, P[f, x_1, \ldots, x_n] \tag{1}$$

[4] All atoms have the form $s \approx t$. Having $\approx$ as the only predicate symbol causes no loss of generality; we model other predicates as function symbols with return sort Bool.

or $\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$, for short, where the second-order variable f represents the function to be synthesized and P is a Σ-formula encoding properties that f must satisfy for each possible input tuple $\boldsymbol{x} = (x_1, \ldots, x_n)$.

In this setting, finding a witness for the satisfiability problem above amounts to finding a function of type $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ in some model of T that satisfies $\forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$. Since we are interested in automatic synthesis, we the restrict ourselves here only to methods that search over a subspace S of solutions representable syntactically as Σ-terms. We will say then that a synthesis conjecture is *solvable* if it has a syntactic solution in S.

Concretely, in this paper we present two approaches that work with classes $\mathbf{L}$ of synthesis conjectures and Σ-theories T that are satisfaction complete wrt $\mathbf{L}$. In both approaches, we solve synthesis conjectures of the form $\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ by relying on quantifier-instantiation techniques for SMT to produce a first-order Σ-term $t[\boldsymbol{x}]$ (of sort σ) such that $\forall \boldsymbol{x}\, P[t, \boldsymbol{x}]$ is T-satisfiable. When this t is found, the synthesized function is the one denoted by $\lambda \boldsymbol{x}.t$.

In principle, to determine the satisfiability of formula (1) an SMT solver supporting the theory T can consider the satisfiability of the (open) formula $\forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ by treating f as an uninterpreted function symbol. This sort of Skolemization is not usually a problem since typical SMT solvers are able to check the T-satisfiability of formulas with uninterpreted symbols. The real challenge is the universal quantification over $\boldsymbol{x}$ because it requires the solver to construct internally (a finite representation of) an interpretation of f that is guaranteed to satisfy $P[f, \boldsymbol{x}]$ for every possible value of $\boldsymbol{x}$ [13,26].

More traditional designs of SMT solvers for handling quantified formulas have focused on instantiation-based methods to show *un*satisfiability. They generate ground instances of quantified formulas until a refutation is found at the ground level [12]. While these techniques are incomplete in general, they have been shown to be quite effective in practice [10,27]. For this reason, we advocate approaches to synthesis geared toward establishing the *unsatisfiability* of the negation of the synthesis conjecture:

$$\forall f\, \exists \boldsymbol{x}\, \neg P[f, \boldsymbol{x}] \tag{2}$$

Thanks to our restriction to satisfaction complete theories, (2) is T-unsatisfiable exactly when the original synthesis conjecture (1) is T-satisfiable.[5] Moreover, as we explain in this paper, a syntactic solution $\lambda x.t$ for (1) can be constructed from a refutation of (2) as opposed to being extracted from the valuation of f in a model of $\forall \boldsymbol{x}\, \neg P[f, \boldsymbol{x}]$.

Proving (2) unsatisfiable poses its own challenge to current SMT solvers, namely, dealing with the second-order universal quantification of f. To our knowledge, no SMT solvers have direct support for higher-order quantification. In the following, however, we describe two specialized methods to refute negated synthesis conjectures like (2) that build on existing capabilities of these solvers.

The first method applies to a restricted, but fairly common, case of synthesis problems $\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ where every occurrence of f in P is in terms of the form $f(\boldsymbol{x})$.

[5] We point out that other approaches in the verification and in the synthesis literature also rely implicitly, and in some cases unwittingly, on this restriction, or stronger ones. We are making satisfaction completeness explicit as a general sufficient condition for reducing satisfiability problems to unsatisfiability ones.

In this case, we can express the problem in the first-order form $\forall \boldsymbol{x}.\exists y.Q[\boldsymbol{x}, y]$, whose negation we can start tackling using appropriate quantifier instantiation techniques.

The second method follows the *syntax-guided synthesis* paradigm [1, 2] where the synthesis conjecture is accompanied by an explicit syntactic restriction on the space of possible solutions. Our syntax-guided synthesis method is based on encoding the syntax of terms as first-order values. We use a deep embedding into an extension of the background theory T with a theory of inductive data types, encoding the restrictions of a syntax-guided synthesis problem.

For the rest of the paper, we fix a Σ-theory T and a class $\mathbf{P}$ of quantifier-free Σ-formulas $P[f, \boldsymbol{x}]$ such that $\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ is a synthesis conjecture and T is satisfaction complete wrt the class $\mathbf{L} := \{\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}] \mid P \in \mathbf{P}\}$ of such conjectures.

3 Refutation-Based Synthesis

When axiomatizing properties of a desired function f of type $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, a particularly well-behaved class are *single-invocation properties* (see, e.g., [15]). These properties include, in particular, standard function contracts, so they can be used to synthesize a function implementation given its postcondition as a relation between the arguments and the result of the function. This is also the form of the specification for synthesis problems considered in complete functional synthesis [18–20], although in our case we assume that the solution exists for all inputs, as opposed to trying to compute the range of the specification relation.

By a single-invocation property we mean a formula $\mathbf{P}$ of the form $\forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ for some quantifier-free formula $Q[\boldsymbol{x}, y]$ that does not contain f. Note that the only occurrences of f in $Q[\boldsymbol{x}, f(\boldsymbol{x})]$ are in subterms of the form $f(\boldsymbol{x})$. The synthesis problem $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ is equivalent in T to the formula

$$\forall \boldsymbol{x}\, \exists y\, Q[\boldsymbol{x}, y] \tag{3}$$

In contrast to (1), formula (3) is first-order. By the semantics of $\forall$ and $\exists$, finding a model $\mathcal{I}$ for it amounts (under the axioms of choice) to finding a function $h : \sigma_1^{\mathcal{I}} \times \cdots \times \sigma_n^{\mathcal{I}} \to \sigma^{\mathcal{I}}$ such that for all $\boldsymbol{s} \in \sigma_1^{\mathcal{I}} \times \cdots \times \sigma_n^{\mathcal{I}}$, the interpretation $\mathcal{I}[\boldsymbol{x} \mapsto \boldsymbol{s}, y \mapsto h(\boldsymbol{s})]$ satisfies $Q[\boldsymbol{x}, y]$. This section describes a general approach for determining the satisfiability of formulas like (3) while computing a syntactic representation of a function like h in the process. For the latter, it will be convenient to assume that the language of functions contains an if-then-else operator ite of type $\mathsf{Bool} \times \sigma \times \sigma \to \sigma$ for each sort σ, with the usual semantics.

If (3) belongs to a fragment that admits quantifier elimination in T, such as the linear fragment of integer arithmetic, determining its satisfiability can be achieved using an efficient method for quantifier elimination [7,23]. Such cases have been examined in the context of software synthesis [19]. Here we propose instead an alternative instantiation-based approach aimed at establishing the unsatisfiability of the *negated* form of (3):

$$\exists \boldsymbol{x}\, \forall y\, \neg Q[\boldsymbol{x}, y] \tag{4}$$

or, equivalently, of a Skolemized version $\forall y\, \neg Q[\mathbf{k}, y]$ of (4) for some tuple $\mathbf{k}$ of fresh uninterpreted constants of the right sort. Finding a T-unsatisfiable finite set Γ of ground

1. $\Gamma := \{\mathsf{G} \Rightarrow Q[\mathbf{k}, \mathsf{e}]\}$
2. Repeat
   If there is a model $\mathcal{I}$ of T satisfying Γ and G then
   $\Gamma := \Gamma \cup \{\neg Q[\mathbf{k}, t[\mathbf{k}]]\}$ for some Σ-term $t[\boldsymbol{x}]$ such that $t[\mathbf{k}]^{\mathcal{I}} = \mathsf{e}^{\mathcal{I}}$;
   otherwise, return "no solution found"
   until Γ contains a T-unsatisfiable set $\{\neg Q[\mathbf{k}, t_1[\mathbf{k}]], \ldots, \neg Q[\mathbf{k}, t_{p+1}[\mathbf{k}]]\}$
3. If $p = 0$ then return $\lambda \boldsymbol{x}.\, t_1[\boldsymbol{x}]$ as a solution;
   otherwise, return $\lambda \boldsymbol{x}.\, \mathsf{ite}(Q[\boldsymbol{x}, t_1[\boldsymbol{x}]], t_1[\boldsymbol{x}],\ (\cdots \mathsf{ite}(Q[\boldsymbol{x}, t_p[\boldsymbol{x}]], t_p[\boldsymbol{x}], t_{p+1}[\boldsymbol{x}])\ \cdots))$

Fig. 1. A refutation-based synthesis procedure for single-invocation property $\exists f \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$.

instances of $\neg Q[\boldsymbol{x}, y]$, which is what an SMT solver would do to prove the unsatisfiability of (4), suffices to solve the original synthesis problem. The reason is that, then, a solution for f can be constructed directly from Γ as shown by the following result.

Proposition 1. Suppose some set $\Gamma = \{\neg Q[\mathbf{k}, t_1[\mathbf{k}]], \ldots, \neg Q[\mathbf{k}, t_{p+1}[\mathbf{k}]]\}$ is T-unsatisfiable, where $t_1[\boldsymbol{x}], \ldots, t_p[\boldsymbol{x}]$ are Σ-terms of sort σ. When $p > 0$, one solution for $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f]$ is $\lambda \boldsymbol{x}.\, \mathsf{ite}(Q[\boldsymbol{x}, t_1[\boldsymbol{x}]], t_1[\boldsymbol{x}],\ (\cdots \mathsf{ite}(Q[\boldsymbol{x}, t_p[\boldsymbol{x}]], t_p[\boldsymbol{x}], t_{p+1}[\boldsymbol{x}])\ \cdots))$; when $p = 0$, one solution is instead $\lambda \boldsymbol{x}.\, t_1[\boldsymbol{x}]$.

Proof: Let ℓ be the solution specified above, and let $\boldsymbol{u}$ be an arbitrary set of ground terms of the same sort as $\boldsymbol{x}$. Given a model $\mathcal{I}$, we show that $\mathcal{I} \models Q[\boldsymbol{u}, \ell(\boldsymbol{u})]$. Consider the case that $\mathcal{I} \models Q[\boldsymbol{u}, t_i[\boldsymbol{u}]]$ for some $i \in \{1, \ldots, p\}$; pick the least such i. Then, $\ell(\boldsymbol{u})^{\mathcal{I}} = (t_i[\boldsymbol{u}])^{\mathcal{I}}$, and thus $\mathcal{I} \models Q[\boldsymbol{u}, \ell(\boldsymbol{u})]$. If no such i exists, then $\mathcal{I} \models \neg Q[\boldsymbol{u}, t_i[\boldsymbol{u}]]$ for all $i = 1, \ldots, p$, and $\ell(\boldsymbol{u})^{\mathcal{I}} = (t_{p+1}[\boldsymbol{u}])^{\mathcal{I}}$, both when $p = 0$ and when $p > 0$. By our assumption and since $\mathbf{k}$ are fresh, we have $\neg Q[\boldsymbol{u}, t_1[\boldsymbol{u}]], \ldots, \neg Q[\boldsymbol{u}, t_p[\boldsymbol{u}]] \models_T Q[\boldsymbol{u}, t_{p+1}[\boldsymbol{u}]]$, which is $Q[\boldsymbol{u}, \ell(\boldsymbol{u})]$. ■

Example 1. Let T be the theory of linear integer arithmetic with the usual signature and integer sort Int. One can show that T is satisfaction complete with respect to the whole class of closed Σ-formulas. Now consider the following property, to be satisfied by a function f of type $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$,

$$P[f, x, y] := f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) \approx x \vee f(x, y) \approx y) \qquad (5)$$

The synthesis problem $\exists f\, \forall x\, y\, P[f, x, y]$ is solved exactly by the function that returns the maximum of its two inputs. As P is a single-invocation property, we can solve that problem by proving the T-unsatisfiability of the conjecture $\exists x\, y\, \forall g\, \neg Q[x, y, g]$ where

$$Q[x, y, g] := g \geq x \wedge g \geq y \wedge (g \approx x \vee g \approx y) \qquad (6)$$

After Skolemization the conjecture becomes $\forall g\, \neg Q[\mathsf{a}, \mathsf{b}, g]$ for fresh constants a and b. When asked to determine the satisfiability of that conjecture an SMT solver may, for instance, instantiate it with a and b for g, producing the T-unsatisfiable set $\{\neg Q[\mathsf{a}, \mathsf{b}, \mathsf{a}], \neg Q[\mathsf{a}, \mathsf{b}, \mathsf{b}]\}$. By Proposition 1, one solution is $\lambda x, y\, \mathsf{ite}(Q[x, y, x], x, y)$, which simplifies to $\lambda x, y.\, \mathsf{ite}(x \geq y, x, y)$, representing the desired $\max$ function. □

Synthesis by Counterexample-Guided Quantifier Instantiation. Given Proposition 1, the main question is how to get the SMT solver to generate the necessary ground

instances from $\forall g\, \neg Q[\mathsf{k}, g]$. Typically, SMT solvers that reason about quantified formulas use heuristic quantifier instantiation techniques based on E-matching [11], which instantiates universal quantifiers with terms occurring in some current set of ground terms built incrementally from the input formula. It is fair to say that E-matching-based heuristic instantiation alone is unlikely to be effective in synthesis, where required terms need to be synthesized based on the semantics of the input specification. This is confirmed by our preliminary experiments, even for simple conjectures. We have developed instead a specialized new technique, which we refer to as *counterexample-guided quantifier instantiation*, that allows the SMT solver to quickly converge in many cases to the instantiations that refute the negated synthesis conjecture.

The new technique is similar to a popular scheme for synthesis known as counterexample-guided inductive synthesis, implemented in various synthesis approaches (e.g., [16, 32]), but with the major difference of being built-in directly into the SMT solver. The technique is illustrated by the procedure in Figure 1, which grows a set Γ of ground instances of $\neg Q[\mathsf{k}, g]$ starting with the formula $\mathsf{G} \Rightarrow Q[\mathsf{k}, \mathsf{e}]$ where G and e are fresh constants of sort Bool and σ, respectively. Intuitively, e represents a current, partial solution for the original synthesis conjecture $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$, while G represents the possibility that the conjecture has a (syntactic) solution in the first place.

The procedure, which may not terminate in general, terminates either when Γ becomes unsatisfiable, in which case it has found a solution, or when all models of Γ falsify G, in which case its has determined that no solution exists. The procedure is not *solution-complete*, that is, guaranteed to return a solution whenever there is one. However, thanks to Proposition 1, it is *solution-sound*: every λ-term it returns is indeed a solution of the original synthesis problem.

Finding instantiations. The choice of the term t in the second step of the procedure is intentionally left underspecified because it can be done in a number of different ways. Having a good heuristic for such instantiations is, however, critical to the effectiveness of the procedure. In a theory with a standard model $\mathcal{I}$ and a standard set of ground Σ-terms denoting the elements of $\sigma^{\mathcal{I}}$, a simple, if naive, choice for t in Figure 1 is the standard term denoting the element $\mathsf{e}^{\mathcal{I}}$. For instance, if σ is Int in the theory of arithmetic, t could be a concrete integer constant $(0, \pm 1, \pm 2, \ldots)$. This choice amounts to testing whether points in the codomain of the sought function f satisfy the original specification P. More sophisticated choices for t, in particular where t contains the variables $\boldsymbol{x}$, may increase the generalization power of this procedure and hence its ability to find a solution. Our present implementation in the CVC4 solver relies on the fact that the model $\mathcal{I}$ in Step 2 is constructed from a set of equivalence classes over terms computed by the solver during its search. The procedure selects the term t among those in the equivalence class of e, other than e itself. For instance, consider formula (5) from the previous example that encodes the single-invocation form of the specification for the max function. The DPLL(T) architecture, on which CVC4 is based, finds a model for $Q[\mathsf{a}, \mathsf{b}, e]$ only if it can first find a subset M of its literals that collectively entail it at the propositional level. Due to the last conjunct, M must include either $e \approx \mathsf{a}$ or $e \approx \mathsf{b}$, and thus whenever a model can be constructed for $Q[\mathsf{a}, \mathsf{b}, e]$, the equivalence class containing e must contain either a or b. Thus using the above selection heuristic, the procedure in Figure 1 will, after at most two iterations of the loop in step 2, add the

$$\forall xy\ \mathsf{ev}(\mathsf{x}, x, y) \approx x \qquad \forall s_1 s_2 xy\ \mathsf{ev}(\mathsf{leq}(s_1, s_2), x, y) \approx (\mathsf{ev}(s_1, x, y) \leq \mathsf{ev}(s_2, x, y))$$
$$\forall xy\ \mathsf{ev}(\mathsf{y}, x, y) \approx y \qquad \forall s_1 s_2 xy\ \mathsf{ev}(\mathsf{eq}(s_1, s_2), x, y) \approx (\mathsf{ev}(s_1, x, y) \approx \mathsf{ev}(s_2, x, y))$$
$$\forall xy\ \mathsf{ev}(\mathsf{zero}, x, y) \approx 0 \qquad \forall c_1 c_2 xy\ \mathsf{ev}(\mathsf{and}(c_1, c_2), x, y) \approx (\mathsf{ev}(c_1, x, y) \wedge \mathsf{ev}(c_2, x, y))$$
$$\forall xy\ \mathsf{ev}(\mathsf{one}, x, y) \approx 1 \qquad \forall c_1 c_2 xy\ \mathsf{ev}(\mathsf{or}(c_1, c_2), x, y) \approx (\mathsf{ev}(c_1, x, y) \vee \mathsf{ev}(c_2, x, y))$$
$$\forall s_1 s_2 xy\ \mathsf{ev}(\mathsf{plus}(s_1, s_2), x, y) \approx \mathsf{ev}(s_1, x, y) + \mathsf{ev}(s_2, x, y)$$
$$\forall s_1 s_2 xy\ \mathsf{ev}(\mathsf{minus}(s_1, s_2), x, y) \approx \mathsf{ev}(s_1, x, y) - \mathsf{ev}(s_2, x, y)$$
$$\forall c s_1 s_2 xy\ \mathsf{ev}(\mathsf{if}(c, s_1, s_2), x, y) \approx \mathsf{ite}(\mathsf{ev}(c, x, y), \mathsf{ev}(s_1, x, y), \mathsf{ev}(s_2, x, y))$$
$$\forall cxy\ \mathsf{ev}(\mathsf{not}(c), x, y) \approx \neg \mathsf{ev}(c, x, y)$$

Fig. 2. Axiomatization of the evaluation operators.

instances $\neg Q[\mathsf{a}, \mathsf{b}, \mathsf{a}]$ and $\neg Q[\mathsf{a}, \mathsf{b}, \mathsf{b}]$ to Γ. As noted in Example 1, these two instances are together T-unsatisfiable.

We expect that more sophisticated instantiation techniques will be incorporated. In particular, both quantifier elimination techniques [8, 24] and approaches currently used to infer invariants from templates [9, 21] are likely to be beneficial for certain classes of synthesis problems. The advantage of developing these techniques within an SMT solver is that they directly benefit both synthesis, and verification in the presence of quantified conjectures, thus fostering cross-fertilization between different fields.

4 Refutation-Based Syntax-Guided Synthesis

In syntax-guided synthesis, the functional specification is strengthened by an accompanying set of syntactic restrictions on the form of the expected solutions. In a recent line of work [1,2,25] these restrictions are expressed by a grammar R (augmented with certain "let expressions") defining the language of solution terms for the synthesis problem. In this section, we present a variant of the approach in the previous section which incorporates the syntactic restriction directly into the SMT solver via a deep embedding of the syntactic restriction R into the solver's logic. The main idea is to encode R by a set of inductive datatypes and build in an interpretation of these datatypes in the solver in terms of the original theory T.

While our approach is parametric in the background theory T and the restriction R, it is best explained with a concrete example. We will use it as a running example for the rest of the section.

Example 2. Consider again the synthesis conjecture (5) from Example 1 but now with a syntactic restriction R for the solution space expressed by these inductive datatypes:

$$\begin{array}{lcl} \mathsf{S} & := & \mathsf{x} \mid \mathsf{y} \mid \mathsf{zero} \mid \mathsf{one} \mid \mathsf{plus}(\mathsf{S}, \mathsf{S}) \mid \mathsf{minus}(\mathsf{S}, \mathsf{S}) \mid \mathsf{if}(\mathsf{C}, \mathsf{S}, \mathsf{S}) \\ \mathsf{C} & := & \mathsf{leq}(\mathsf{S}, \mathsf{S}) \mid \mathsf{eq}(\mathsf{S}, \mathsf{S}) \mid \mathsf{and}(\mathsf{C}, \mathsf{C}) \mid \mathsf{or}(\mathsf{C}, \mathsf{C}) \mid \mathsf{not}(\mathsf{C}) \end{array}$$

The datatypes are meant to encode a term signature that includes nullary constructors for the variables x, y of (5), and constructors for the symbols of the arithmetic theory T. Terms of sort S (resp., C) refer to theory terms of sort Int (resp., Bool).

Instead of the theory of linear integer arithmetic, we now consider its combination T_{D} with the theory of the datatypes above extended with two *evaluation operators*, that is, two function symbols $\mathsf{ev}^{\mathsf{S}\times\mathsf{Int}\times\mathsf{Int}\to\mathsf{Int}}$ and $\mathsf{ev}^{\mathsf{C}\times\mathsf{Int}\times\mathsf{Int}\to\mathsf{Bool}}$ respectively embedding

Repeat
1. If there is a model $\mathcal{I}$ of T_{D} satisfying Γ and G then
   $\Gamma := \Gamma \cup \{\neg Q[\mathsf{e}^{\mathcal{I}}, \mathbf{k}]\}$ where $\mathbf{k}$ are fresh constants;
   otherwise, return "no solution found"
2. If there is a model $\mathcal{J}$ of T_{D} satisfying Γ then
   $\Gamma := \Gamma \cup \{\mathsf{G} \Rightarrow Q[\mathsf{e}, \mathbf{k}^{\mathcal{J}}]\}$ where $\mathbf{k}$ are the constants in Step 1;
   otherwise, return $e^{\mathcal{I}}$ as a solution

Fig. 3. A refutation-based syntax-guided synthesis procedure for $\exists f\, \forall \boldsymbol{x}\, Q[f, \boldsymbol{x}]$.

S in Int and C in Bool. We define T_{D} so that all of its models satisfy the formulas in Figure 2. The evaluation operators effectively define an interpreter for programs (i.e., terms of sort S and C) with input parameters x and y.

It is possible to instrument an SMT solver that supports user-defined datatypes, quantifiers and linear arithmetic so that it constructs automatically from the syntactic restriction R both the datatypes S and C and the two evaluation operators. Reasoning about S and C is done by the built-in subsolver for datatypes. Reasoning about the evaluation operators is achieved by reducing ground terms of the form $\mathsf{ev}(d, v, u)$ to smaller terms by mean of selected instantiations of the axioms from Figure 2, with a number of instantiations proportional to the size of d. It is also possible to show that T_{D} is satisfaction complete wrt the class $\mathbf{L}_2 := \{\exists g\, \forall xy\, P[\lambda uv.\, \mathsf{ev}(d, u, v), x, y] \mid P \in \mathbf{P}\}$ where instead of terms of the form $f(t_1, t_2)$ in P we have, modulo β-reductions, terms of the form $\mathsf{ev}(d, t_1, t_2)$.[6] The original conjecture (5) can then be restated as in T_{D} as

$$Q[g, x, y] := \mathsf{ev}(g, x, y) \geq x \wedge \mathsf{ev}(g, x, y) \geq y \wedge (\mathsf{ev}(g, x, y) \approx x \vee \mathsf{ev}(g, x, y) \approx y)$$

where g is a variable of type S. Note that, in contrast to (5), this problem is first-order, because quantification over functions is replaced by quantification over the inductive data types that represent the syntax of this function.

When asked for a solution for (5) under the restriction R, the instrumented SMT solver will try to determine instead the T_{D}-unsatisfiability of $\forall g\, \exists xy\, \neg Q[g, x, y]$. Instantiating g in the latter formula with $s = \mathsf{if}(\mathsf{leq}(\mathsf{x}, \mathsf{y}), \mathsf{y}, \mathsf{x})$, say, produces the formula $\exists xy\, \neg Q[s, x, y]$ which the solver can prove to be T_{D}-unsatisfiable. This suffices to show that the term $\mathsf{ite}(x \leq y, y, x)$, the analog of s in the language of T, is a solution of the synthesis conjecture (5) under the syntactic restriction R. □

To produce a proof of the unsatisfiability of formulas like $\forall g\, \exists \boldsymbol{x}\, \neg Q[g, \boldsymbol{x}]$ where g ranges over the datatype that encodes the restricted solution space, we use a procedure similar to that in Section 3, but specialized to the extended theory T_{D}. The procedure is described in Figure 3. Like the one in Figure 1, it uses an uninterpreted constant e representing a solution candidate, and a Boolean variable G representing the existence of a solution. The main difference, of course, is that now e ranges over the datatype that encodes the original sort. In any model of T_{D}, a term of datatype sort evaluates to a term built exclusively with constructor symbols. This is why the procedure returns the

[6] We stress again, that both the instrumentation of the solver and the satisfaction completeness argument for the extended theory are generic with respect to the syntactic restriction on the synthesis problem and the original satisfaction complete theory T.

Step	Model	Added Formula
1	$\{\mathsf{e} \mapsto \mathsf{x}, \ldots\}$	$\neg Q[\mathsf{x}, \mathsf{a}_1, \mathsf{b}_1]$
2	$\{\mathsf{a}_1 \mapsto 0, \mathsf{b}_1 \mapsto 1, \ldots\}$	$\mathsf{G} \Rightarrow Q[\mathsf{e}, 0, 1]$
1	$\{\mathsf{e} \mapsto \mathsf{y}, \ldots\}$	$\neg Q[\mathsf{y}, \mathsf{a}_2, \mathsf{b}_2]$
2	$\{\mathsf{a}_2 \mapsto 1, \mathsf{b}_2 \mapsto 0, \ldots\}$	$\mathsf{G} \Rightarrow Q[\mathsf{e}, 1, 0]$
1	$\{\mathsf{e} \mapsto \mathsf{one}, \ldots\}$	$\neg Q[\mathsf{one}, \mathsf{a}_3, \mathsf{b}_3]$
2	$\{\mathsf{a}_3 \mapsto 2, \mathsf{b}_3 \mapsto 0, \ldots\}$	$\mathsf{G} \Rightarrow Q[\mathsf{e}, 2, 0]$
1	$\{\mathsf{e} \mapsto \mathsf{plus}(\mathsf{x}, \mathsf{y}), \ldots\}$	$\neg Q[\mathsf{plus}(\mathsf{x}, \mathsf{y}), \mathsf{a}_4, \mathsf{b}_4]$
2	$\{\mathsf{a}_4 \mapsto 1, \mathsf{b}_4 \mapsto 1, \ldots\}$	$\mathsf{G} \Rightarrow Q[\mathsf{e}, 1, 1]$
1	$\{\mathsf{e} \mapsto \mathsf{if}(\mathsf{leq}(\mathsf{x}, \mathsf{one}), \mathsf{one}, \mathsf{x}), \ldots\}$	$\neg Q[\mathsf{if}(\mathsf{leq}(\mathsf{x}, \mathsf{one}), \mathsf{one}, \mathsf{x}), \mathsf{a}_5, \mathsf{b}_5]$
2	$\{\mathsf{a}_5 \mapsto 1, \mathsf{b}_5 \mapsto 2, \ldots\}$	$\mathsf{G} \Rightarrow Q[\mathsf{e}, 1, 2]$
1	$\{\mathsf{e} \mapsto \mathsf{if}(\mathsf{leq}(\mathsf{x}, \mathsf{y}), \mathsf{y}, \mathsf{x}), \ldots\}$	$\neg Q[\mathsf{if}(\mathsf{leq}(\mathsf{x}, \mathsf{y}), \mathsf{y}, \mathsf{x}), \mathsf{a}_6, \mathsf{b}_6]$
2	none	

Fig. 4. A run of the procedure from Figure 3.

value of e in the model $\mathcal{I}$ found in Step 1. As we saw in the example, a solution for the original problem can then be reconstructed for the returned datatype term.

Lemma 1. The procedure maintains the following invariants:

1. In Step 1, Γ is T_{D}-satisfiable.
2. In Step 2, if Γ is T_{D}-satisfiable, it has a model that satisfies G.

The procedure is implemented in CVC4. Figure 4 shows a run of that implementation over the conjecture from Example 2. In this run, note that each model found for e satisfies all values of counterexamples found for previous candidates. After the sixth iteration of Step 1, the procedure finds the candidate $\mathsf{if}(\mathsf{leq}(\mathsf{x}, \mathsf{y}), \mathsf{y}, \mathsf{x})$, for which no counterexample exists, indicating that the procedure has found a solution for the synthesis conjecture. Currently, this problem can be solved in about 0.5 seconds in the latest development version of CVC4.

Fairness. In practice, a necessary technique for limiting the candidate programs is to consider smaller programs before larger ones. Adapting techniques for finding finite models of minimal size [28], we use a strategy that searches for programs of size 1 only after we have exhausted the search for programs of size 0. In solvers based, like CVC4, on the DPLL(T) architecture, this can be accomplished by introducing a splitting lemma of the form $(\mathsf{size}(\mathsf{e}) \leq 0 \vee \neg \mathsf{size}(\mathsf{e}) \leq 0)$, and asserting $\mathsf{size}(\mathsf{e}) \leq 0$ as the first decision literal, where size is a function symbol of type $\sigma \to \mathsf{Int}$ for every datatype sort σ standing for the function that maps each datatype value to its term size (i.e., the number of non-nullary constructor applications in the term). We do the same for $\mathsf{size}(\mathsf{e}) \leq 1$ if and when $\neg \mathsf{size}(\mathsf{e}) \leq 0$ becomes asserted. We have extended the decision procedure for inductive datatypes in CVC4 [6] to handle constraints involving size. The extended procedure remains a decision procedure for input problems with a concrete upper bound on the terms of the form $\mathsf{size}(u)$ for each variable or uninterpreted constant u of a datatype sort in the problem. This is enough for our purposes since the only such variable or constant in our synthesis procedure is e.

Proposition 2. Using the fairness strategy above, the procedure in Figure 3 has the following properties:

1. (Solution Soundness) Every term it returns can be mapped to a solution of the original synthesis conjecture $\exists f \, \forall \boldsymbol{x} \, P[f, \boldsymbol{x}]$ under the restriction R.
2. (Refutation Soundness) If it answers "no solution found", the original conjecture has no solutions under the restriction R.
3. (Solution Completeness) If the original conjecture has a solution under R, the procedure will find one.

The procedure is solution sound for a similar reason as noted in the proof of Proposition 1, noting that the unsatisfiable core of Γ contains only one instance of Q as added in step 1 when it terminates with a solution. Second, the procedure is refutation sound since when no model satisfies G, we have that even an arbitrary e cannot satisfy the current set of instances we have added to Γ in step 2. It is possible that the procedure does not terminate, but only if there are no solutions. Finally, the procedure is solution complete because any single run of step 1 or 2 is terminating since our background theory T_{D} is decidable. Each invocation of step 1 is guaranteed to produce a new candidate since T_{D} is also satisfaction complete. Thus, in the worst case, the procedure amounts to fairly enumerating and testing a stream of candidate programs until a solution is found.

5 Single Invocation Techniques for Syntax-Guided Problems

Given a set R of syntactic restrictions expressed by an inductive datatype S, consider the case when both S contains the constructor $\mathsf{if} : \mathsf{C} \times \mathsf{S} \times \mathsf{S} \to \mathsf{S}$ for some inductive datatype C, and the property of the function we are synthesizing is single-invocation and can be expressed as a term of sort C. For instance, the property from Example 2 can be phrased as:

$$Q[g, u, v] := \mathsf{ev}(\mathsf{and}(\mathsf{leq}(\mathsf{x}, g), \mathsf{and}(\mathsf{leq}(\mathsf{y}, g), \mathsf{or}(\mathsf{eq}(g, \mathsf{x}), \mathsf{eq}(g, \mathsf{y})))), u, v) \quad (7)$$

where g has sort S and k_1 and k_2 have sort Int. The procedure in Figure 1 is applicable to the conjecture $\exists uv \, \forall g \, \neg Q(g, u, v)$ since it emits solutions meeting our syntactic requirements. Running this procedure on this form of the conjecture has the advantage over the procedure in Figure 3 that only the outputs of a solution need to be synthesized, and not conditions in ite-terms.

However, our evaluation has found that the overhead of using an embedding into datatypes and evaluation operators for syntax-guided problems is significant with respect to the performance of the solver on problems with no syntactic restrictions. For this reason, in this section we advocate an approach for single invocation synthesis conjectures with syntactic restrictions that runs the procedure from Figure 1 while ignoring the syntactic restrictions R for solutions, and when successful in generating a solution, subsequently reconstructs from that solution one that meets the requirements R.

In more detail, say the procedure from Figure 1 returns a solution $\lambda \boldsymbol{x}.t$ for a function f that has syntactic restrictions R. To reconstruct solution $\lambda \boldsymbol{x}.t$ that meets the requirements of R, we first apply rewriting techniques to reduce the size and complexity of t, including ite simplification, redundant subterm elimination and destructive equality resolution. We then traverse the structure of t in a top-down manner to find a set $\mathcal{U}$ of pairs of the form $(f(\boldsymbol{u}), S)$, where $f(\boldsymbol{u})$ is a subterm of t that, according to R and its

rcon(t, S, A) :
If $(t, S) \in A$, return $\{\}$.
Otherwise, choose a $f(t_1, \ldots, t_n)$ such that $f(t_1, \ldots, t_n)\downarrow = t$ and $\mathsf{f}^{S_1 \ldots S_n S} \in S$, and return $\bigcup_{i \in \{1,\ldots,n\}} \mathbf{rcon}(t_i, S_i, A)$.
If none exists, return $\{(t, S)\}$.

Let $A = \{\}$. Repeat, for $i = 1, 2, \ldots$
- Let $\mathcal{U} = \mathbf{rcon}(t, \mathsf{S}, A)$. If $\mathcal{U}$ is empty, return "success".
- For each datatype S_j occurring in $\mathcal{U}$,
  - Let d_i be the i^{th} term in a fair enumeration of S_j.
  - Let t_i be the analog of d_i in the background theory.
  - Add $(t_i\downarrow, S_j)$ to A.

Fig. 5. A procedure for finding a term of datatype S whose analog is equivalent to t. The subprocedure **rcon** returns a set of pairs of the form (t_i, S_i) representing that it remains to find a term of datatype S_i whose analog is equivalent to t_i. We write $t_i\downarrow$ to denote the normal form of t_i. When this set of pairs is empty, the procedure returns successfully, and the solution for t can be constructed by traversing the structure of t while referencing the pairs from A when needed.

placement within t, must be an application of an operator in S, and where $f \notin S$. For example, given the grammar from Example 2 and the solution $\lambda x, y.\, x + 2 * y$, then $\mathcal{U}$ is the set $\{(2{*}y, \mathsf{S})\}$, noting that S does not contain multiplication. Then, for each datatype T in our grammar R required for reconstructing t, we enumerate terms $d_1, \ldots, d_n$ of type T. If we find a d_i that corresponds to the (built-in) term t_i that is theory-equivalent to some subterm t' of t, we remove from $\mathcal{U}$ a set of pairs corresponding to the subterms of t' we have yet to successfully reconstruct. Reconstruction succeeds when the set $\mathcal{U}$ is empty. For instance, our reconstruction would succeed for $\lambda xy.\, x + 2 * y$ when the term $\mathsf{plus}(y, y)$ was enumerated, which corresponds to the term $y + y$, and is equivalent to $2 * y$. This tells us that $\lambda xy.\, x + (y + y)$ is a solution for the conjecture meeting the syntactic requirements R. Here, the equivalence relation between expressions can be easily underapproximated by CVC4 due to the common use of a *normal form* for terms (we write $t\downarrow$ to denote the normal form of term t), where two terms are determined to be equivalent if and only if they have the same normal form. The normal form for ground terms can vary depending on the background theory and its implementation. As such, subterms of our solution t (which are in a normal form) often do not meet the requirements of our grammar R, making the techniques described above necessary for the vast majority of solutions we considered for reconstruction.

A simplified version of this procedure is given in Figure 5. In the implementation, $\mathcal{U}$ is represented as a directed acyclic graph (dag) whose nodes are pairs (t, S) for term t and datatype S, and whose edges contain links to the direct subchildren of that term. Datatype terms are enumerated for all types that are connected to unreconstructed terms in this dag, which is incrementally pruned as pairs are added to A until it becomes empty. Although the overhead of this procedure can be significant in cases where large subterms do not meet syntactic restrictions from R, we found that in practice it terminated successfully in a short amount of time for a majority of the solutions we considered where reconstruction was possible, which we discuss in the next section. Furthermore, it makes our implementation more robust to the manner in which proper-

ties are specified, that is, properties that are equivalent modulo normalization (which is parametric in the built-in theories we consider) are effectively treated as the same.

6 Experimental Evaluation

We implemented the techniques from the previous sections in the SMT solver CVC4 [4], which has support for quantified formulas and a wide range of theories including arithmetic, bitvectors, and inductive datatypes. We evaluated our implementation on 243 benchmarks used in the SyGuS 2014 competition [1] that were publicly available on StarExec. The benchmarks are in a new format for specifying syntax-guided synthesis problems [25]. All benchmarks in this library contain synthesis conjectures whose background theory is either quantifier-free linear integer arithmetic or bitvectors. Parsing support has been added to CVC4 for most features of this format. We made some minor modifications to SyGuS benchmarks to avoid naming conflicts, and to explicitly define several bitvector operators that are not supported natively by CVC4. Our results compare CVC4 primarily against the Enumerative CEGIS solver **ESolver**, the winner of the SyGuS 2014 competition. In our testing, we found that **ESolver** performed significantly better than the other entrants of this competition.

We ran multiple configurations of CVC4 to measure the strength of the techniques mentioned in this paper. First, the configuration **cvc4+sg** ran the syntax-guided algorithm from Section 4 only, even in cases where the benchmark contained a conjecture whose property was single invocation. Second, the configuration **cvc4+si-r** ran the algorithm from Section 3 on all benchmarks having conjectures that it could deduce were single invocation. In total, it deduced that 176 of the 243 benchmarks could be rewritten into a form that was single invocation. This configuration was designed to measure the number of solutions generated by the procedure from Figure 1, which indeed are solutions for the conjecture in question, but do not necessarily reside in the required grammar. Finally, the configuration **cvc4+si** measured the number of benchmarks that could be solved by **cvc4+si-r** *and* whose solution could be reconstructed as a term in required syntax, as described in the Section 5. All configurations on all benchmarks were run on the StarExec cluster. [7]

	array (32)		**bv** (7)		**hd** (56)		**icfp** (50)		**int** (15)		**let** (8)		**multf** (8)		**Total** (176)	
	#	time	#	time	#	time	#	time	#	time	#	time	#	time	#	time
ESolver	3	467.6	2	71.6	50	888	0	0	5	1380.4	2	0.1	7	0.6	69	2808.3
cvc4+sg	2	1415.3	0	0	33	1822.6	1	0.5	3	1.7	2	0.8	7	647.9	48	3888.8
cvc4+si-r	(32)	1.2	(6)	137.9	(56)	2.1	(43)	3340.7	(15)	0.6	(7)	1.3	(7)	0.1	(166)	3483.9
cvc4+si	30	798.8	3	0.04	50	1754.7	0	0	6	0.2	1	0.7	7	0.1	97	2554.5

Fig. 6. Results for single invocation properties, showing number of benchmarks solved by each solver and configuration over 8 benchmark classes with a 1800s timeout. The number of benchmarks solved by configuration **cvc4+si-r** are shown in parentheses, since the solutions it produced do not necessarily meet syntactic restrictions.

[7] The results can be found at `https://www.starexec.org/starexec/secure/details/job.jsp?id=6561` and `https://www.starexec.org/starexec/secure/details/job.jsp?id=6440`. A detailed summary can be found at `http://lara.epfl.ch/~reynolds/CAV2015-synth`.

The results for benchmarks with single invocation properties are shown in Figure 6. Configuration **cvc4+si-r** found a solution (although not necessarily residing in the required grammar) for a majority of benchmarks very quickly, as it terminates successfully for 166 of 176 benchmarks, and for 155 of these cases it terminates in less than a second. Not all solutions found using this method meet the grammar specification. Nevertheless, our methods for reconstructing these solutions into the required grammar succeeded in 58% of these cases, as the configuration **cvc4+si** was able to reconstruct 97 of these solutions into one that met the requirements for the grammar. This number is 28 higher than the number solved by **ESolver** (the best known solver for these benchmarks), which solves 69. In total, **cvc4+si** solved 34 benchmarks that **ESolver** did not, while **ESolver** solves 6 that **cvc4+si** did not.

By ignoring the single invocation aspect of the benchmarks, the configuration **cvc4+sg** solves 48. Regardless, **cvc4+sg** was able to solve a small number of benchmarks that could not be solved by any other configuration, including one from the **icfp** class whose solution was a single argument function over bitvectors that shifted its input right by four bits. We also comment that in addition to being solution complete, the configuration **cvc4+sg** produces solutions that are guaranteed to be of minimal term size, which the other configurations do not guarantee. Of the 47 benchmarks solved by both **cvc4+sg** and **ESolver**, we found that in 6 cases, the solution returned by **cvc4+sg** had a smaller term size than **ESolver**, and in no cases did **ESolver** return a smaller solution than **cvc4+sg**. This provides confirming evidence that the fairness techniques for term size from Section 4 ensure minimal size solutions.

	2	3	4	5	6	7	8	9	10
Esolver	0.01	1377.10	–	–	–	–	–	–	–
cvc4+si	0.01	0.02	0.03	0.05	0.1	0.3	1.6	8.9	81.5

Fig. 7. Results for parametric benchmarks class encoding the maximum of n integers. The columns show the run time for **ESolver** and CVC4 with a 3600s timeout.

We remark that solutions returned by **cvc4+si-r** are often large, often having the order of 10k subterms for harder benchmarks. However, after exhaustively applying simplification techniques during reconstruction with the configuration **cvc4+si**, we found that the size of solutions is comparable to other solvers, and in some cases even smaller. When comparing the term size of solutions produced by **cvc4+si** against the solutions produced by **ESolver** in the 114 benchmarks they both solve, in 15 cases **ESolver** produced a smaller solution, whereas in 9 cases **cvc4+si** produced the smaller solution. Only in 2 cases did **cvc4+si** produce a solution that had 10 more subterms that the solution produced by **ESolver**. This indicates that in addition to having a high precision, the simplification techniques from Section 5 used while reconstructing solutions are effective at producing succinct solutions for this benchmark library as well.

Competitive advantage on single-invocation properties in the presence of ite. For certain benchmarks and classes, the performance of **cvc4+si** was significantly more scalable than state-of-the-art synthesis tools. For instance, we show in Figure 7 how much better CVC4 scales for instances of the max example for finding the maximum of n integers against the performance of **ESolver**. As reported in [1], *no solver* in the competition was able to synthesize a function to compute the maximum of 5 integers in a one hour timeout. For benchmarks from the **array** class of benchmarks, whose

	int (3)		**invgu** (28)		**invg** (28)		**vctrl** (8)		**Total** (67)	
	#	time	#	time	#	time	#	time	#	time
ESolver	3	1.6	25	86.5	25	85.3	5	29.4	58	202.8
cvc4+sg	3	1347.9	23	1172.5	22	2231.8	5	1075.8	53	5828.0

Fig. 8. Results for properties that are not single invocation, showing number of benchmarks solved by CVC4 and **ESolver** over 4 benchmark classes with a 1800s timeout.

solutions are loop-free programs that compute the first instance of an element in a sorted array, the best reported solver for these in [1] was Sketch, which solved a problem for an array of length 7 in approximately 30 minutes.[8] In contrast, **cvc4+si** was able to reconstruct solutions for arrays of size 15 (the largest benchmark in the class) within 0.3 seconds, and solved all but 8 of the benchmarks over the entire class each within 1 second.

For benchmarks that had conjectures that were not single invocation properties, CVC4 must resort to the algorithm described in Figure 3. The results for **ESolver** and **cvc4+sg** on these benchmarks are shown in Figure 8, showing that **cvc4+sg** solves 53 while **ESolver** solves 58 (it additionally reports that 6 have no solutions). In more detail, ESolver solved 7 benchmarks that CVC4 did not, while CVC4 solved 2 benchmarks (from the **vctrl** class) that ESolver did not. In terms of precision, CVC4 is somewhat competitive with the state of the art for non-single invocation properties. To give other points of comparison, the 2014 SyGuS competition [1] reports that the second best solver (the stochastic solver) solved 40 of these 67 benchmarks within a one hour timeout, and Sketch solved 23 of 67 within a one hour timeout.

In total, over all benchmarks, 150 benchmarks can be solved by a configuration of CVC4 that, whenever possible, runs the methods for single invocation properties (Figure 1), and otherwise runs the method described in Section 4. This number is 23 higher than the 127 benchmarks solved by ESolver over all SyGuS benchmarks. Running both configuration **cvc4+sg** and **cvc4+si** in parallel solves 153 benchmarks, meaning that CVC4 is highly competitive with state-of-the-art tools for syntax guided synthesis. Our performance is noticeably better than ESolver on single invocation properties, where our new quantifier instantiation techniques give CVC4 a distinct advantage.

7 Conclusion

We have shown that SMT solvers can be not only used as subroutines for synthesis tasks, but that they can perform synthesis themselves. To the best of our knowledge, this is the first implementation of synthesis inside an SMT solver, and it already shows promise. The solutions being solved by the broad class of syntax-guided synthesis problems can be expressed in our framework. Using a dedicated quantifier instantiation technique and enumeration features of the theory of inductive data types, our implementation is competitive with the state of the art. Moreover, for the important class of single-invocation properties when syntax permits if-then-else, our implementation significantly outperforms solvers that participated in the 2014 syntax-guided competition.

[8] These benchmarks, as contributed to the SyGuS benchmark set, use integer variables only; they were generated by expanding fixed-size arrays but do not refer to actual arrays.

References

1. R. Alur, R. Bodik, 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. To Appear in Marktoberdrof NATO proceedings, 2014. `http://sygus.seas.upenn.edu/files/sygus_extended.pdf`, retrieved 2015-02-06.
2. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
3. R. Alur, M. M. K. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa. Synthesizing finite-state protocols from scenarios and requirements. In E. Yahav, editor, *Haifa Verification Conference*, volume 8855 of *LNCS*, pages 75–91. Springer, 2014.
4. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of CAV'11*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
5. C. Barrett, M. Deters, L. M. de Moura, A. Oliveras, and A. Stump. 6 years of SMT-COMP. *JAR*, 50(3):243–277, 2013.
6. C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.
7. N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin Heidelberg, 2010.
8. N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *LNCS*, pages 316–330. Springer, 2010.
9. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In R. Cousot, editor, *VMCAI*, volume 3385 of *LNCS*, pages 1–24. Springer, 2005.
10. L. de Moura and N. Bjørner. Efficient E-Matching for SMT solvers. In *CADE, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.
11. L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
12. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.
13. Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
14. C. C. Green. Application of theorem proving to problem solving. In D. E. Walker and L. M. Norton, editors, *IJCAI*, pages 219–240. William Kaufmann, 1969.
15. S. Jacobs and V. Kuncak. Towards complete reasoning about axiomatic specifications. In *Verification, Model Checking, And Abstract Interpretation*, pages 278–293. Springer Berlin Heidelberg, 2011.
16. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE*, pages 215–224. ACM, 2010.
17. E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *OOPSLA*, pages 407–426. ACM, 2013.

18. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 316–329. ACM, 2010.
19. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *CACM*, 55(2):103–111, 2012.
20. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Functional synthesis for linear arithmetic and sets. *STTT*, 15(5-6):455–474, 2013.
21. R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In *Computer Aided Verification (CAV)*, 2014.
22. Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *TOPLAS*, 2(1):90–121, 1980.
23. D. Monniaux. Quantifier elimination by lazy model enumeration. In *Computer Aided Verification*, pages 585–599. Springer Berlin Heidelberg, 2010.
24. D. Monniaux. Quantifier elimination by lazy model enumeration. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.
25. M. Raghothaman and A. Udupa. Language to specify syntax-guided synthesis problems. *CoRR*, abs/1405.5590, 2014.
26. A. Reynolds, C. Tinelli, A. Goel, S. Krstić, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In M. P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction (Lake Placid, NY, USA)*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.
27. A. Reynolds, C. Tinelli, and L. D. Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2014.
28. A. J. Reynolds. *Finite Model Finding in Satisfiability Modulo Theories*. PhD thesis, The University of Iowa, 2013.
29. L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In J. Flinn and H. Levy, editors, *OSDI*, pages 661–676. USENIX Association, 2014.
30. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
31. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *ASPLOS*, pages 404–415. ACM, 2006.
32. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.
33. S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.