Counterexample-Guided Quantifier Instantiation for Synthesis in SMT[⋆] [⋆⋆]

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

[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. We introduce the first program synthesis engine implemented inside an SMT solver. We present an approach that extracts solution functions from unsatisfiability proofs of the negated form of synthesis conjectures. We also discuss novel counterexample-guided techniques for quantifier instantiation that we use to make finding such proofs practically feasible. 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 form. As an alternative, we show how to use evaluation function axioms to embed syntactic restrictions into constraints over algebraic datatypes, and then use an algebraic datatype decision procedure to drive synthesis. Our experimental evaluation on syntax-guided synthesis benchmarks shows that our implementation in the CVC4 SMT solver is competitive with state-of-the-art tools for synthesis.

1 Introduction

The synthesis of functions that meet a given specification is a long-standing fundamental goal that has received great attention recently. This functionality directly applies to the synthesis of functional programs [17, 18] but also translates to imperative programs through techniques that include bounding input space, verification condition generation, and invariant discovery [28–30]. Function synthesis is also an important subtask in the synthesis of protocols and reactive systems, especially when these systems are infinite-state [3, 27]. The SyGuS format and competition [1, 2, 22] 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 [12, 20]. It is notable that early research [20] 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

[⋆] This work is supported in part by the European Research Council (ERC) Project *Implicit Programming* and Swiss National Science Foundation Grant *Constraint Solving Infrastructure for Program Analysis*.

[⋆⋆] This paper is dedicated to the memory of Morgan Deters who died unexpectedly in Jan 2015.

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 or 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 [15] 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 existing algorithms and introduce new ones to make SMT-based synthesis competitive. Specifically, we do the following.

- We show how to formulate an important class of synthesis problems as the problem of disproving universally quantified formulas, and how to synthesize functions automatically from selected instances of these formulas.
- We present counterexample-guided techniques for quantifier instantiation, which are crucial to obtain competitive performance on synthesis tasks.
- We discuss techniques to simplify the synthesized functions, to help ensure that they are small and adhere to specified syntactic requirements.
- We show how to encode syntactic restrictions using theories of algebraic datatypes and axiomatizable evaluation functions.
- We show that for an important class of single-invocation properties, the synthesis of functions from relations, the implementation of our approach in CVC4 significantly outperforms leading tools from the SyGuS competition.

Preliminaries. 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. We fix a set $\mathbf{S}$ of *sort symbols* and an (infix) equality predicate $\approx$ of type $\sigma \times \sigma$ for each $\sigma \in \mathbf{S}$. 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* $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$. For each sort σ we identity the type $() \rightarrow \sigma$ 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* $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$, where $n \geq 0$ and $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma^{\mathrm{s}}$. 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 (Σ-)term, atom, literal, clause, and formula with variables in $\mathbf{X}$ are defined as usual in second-order logic. All atoms have the form $s \approx t$. Having $\approx$ as the only predicate symbol causes no loss of generality since we can model other predicate symbols as function symbols with return sort Bool. We will, however, write just t in place of the atom $t \approx \top$, to simplify the notation. 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 $\boldsymbol{x} = (x_1, \ldots, x_n)$ has no repeated variables, we write $e[\boldsymbol{x}]$ to denote that all of e's free variables are from $\boldsymbol{x}$; if $\boldsymbol{t} = (t_1, \ldots, t_n)$ is a term tuple, we write $e[\boldsymbol{t}]$ for the term or formula obtained from e by simultaneously replacing, for all $i = 1, \ldots, n$, every 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 $u^{\sigma_1 \cdots \sigma_n \sigma} \in \mathbf{X} \cup \Sigma^{\mathrm{f}}$ 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 $n = 0$. The interpretation $\mathcal{I}$ induces as usual a mapping from terms t of sort σ to elements $t^{\mathcal{I}}$ of $\sigma^{\mathcal{I}}$. 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 fragment $\mathbf{L}$ of the language of Σ-formulas, a Σ-theory T is *satisfaction complete with respect 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, reals, strings, algebraic datatypes, 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 with respect to considerably more restricted classes of formulas.

2 Synthesis inside an SMT Solver

We are interested in synthesizing computable functions automatically from formal logical specifications stating properties of these functions. As we show later, under the right conditions, we can formulate a version of the synthesis problem in *first-order logic* alone, which allows us to tackle the problem using SMT solvers.

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 consider *synthesis conjectures* expressed as (well-sorted) formulas of the form

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

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 all possible values of the input tuple $\boldsymbol{x} = (x_1, \ldots, x_n)$. In this setting, finding a witness for this satisfiability problem 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 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.

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 a synthesis conjecture $\exists f\,\forall \boldsymbol{x}\,P[f, \boldsymbol{x}]$ by relying on quantifier-instantiation techniques 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 denoted by $\lambda \boldsymbol{x}.\,t$.

In principle, to determine the satisfiability of $\exists f\,\forall \boldsymbol{x}\,P[f, \boldsymbol{x}]$ 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 for SMT solvers as many of them can process 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 $\boldsymbol{x}$ [11, 24].

More traditional SMT solver designs to handle universally quantified formulas have focused on instantiation-based methods to show *un*satisfiability. They generate ground instances of those formulas until a refutation is found at the ground level [10]. While these techniques are incomplete in general, they have been shown to be quite effective in practice [9, 25]. 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.[4] 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 $\forall \boldsymbol{x}\,P[f, \boldsymbol{x}]$.

Two synthesis methods. 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 so far had 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 $f(\boldsymbol{x})$. In this case, we can express the problem in the first-order form $\forall \boldsymbol{x}.\exists y.Q[\boldsymbol{x}, y]$ and then tackle its negation 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 algebraic data types, encoding the restrictions of a syntax-guided synthesis problem.

[4] Other approaches in the verification and synthesis literature also rely implicitly, and in some cases unwittingly, on this restriction or stronger ones. We make satisfaction completeness explicit here as a sufficient condition for reducing satisfiability problems to unsatisfiability ones.

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 T is satisfaction complete with respect to the class of synthesis conjectures $\mathbf{L} := \{\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}] \mid P \in \mathbf{P}\}$.

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., [13]). 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 [16–18]. Note that, in our case, we aim to prove that the output exists for all inputs, as opposed to, more generally, computing the set of inputs for which the output exists.

A *single-invocation property* is any formula of the form $Q[\boldsymbol{x}, f(\boldsymbol{x})]$ obtained as an instance of a quantifier-free formula $Q[\boldsymbol{x}, y]$ not containing f. Note that the only occurrences of f in $Q[\boldsymbol{x}, f(\boldsymbol{x})]$ are in subterms of the form $f(\boldsymbol{x})$ with the *same* tuple $\boldsymbol{x}$ of *pairwise distinct* variables.[5] The conjecture $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ is logically equivalent to the *first-order* formula

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

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 considers the case when $\mathbf{P}$ consists of single-invocation properties and 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 [7,21]. Such cases have been examined in the context of software synthesis [17]. 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 instances of $\neg Q[\boldsymbol{k}, y]$, 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 indicated by the following result.

Proposition 1. Suppose some set $\Gamma = \{\neg Q[\mathbf{k}, t_1[\mathbf{k}]], \ldots, \neg Q[\mathbf{k}, t_p[\mathbf{k}]]\}$ where $t_1[\boldsymbol{x}]$, …, $t_p[\boldsymbol{x}]$ are Σ-terms of sort σ is T-unsatisfiable. One solution for $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ is $\lambda \boldsymbol{x}.\, \mathsf{ite}(Q[\boldsymbol{x}, t_p], t_p, (\cdots\, \mathsf{ite}(Q[\boldsymbol{x}, t_2], t_2, t_1)\, \cdots))$.

[5] An example of a property that is *not* single-invocation is $\forall x_1\, x_2\, f(x_1, x_2) \approx f(x_2, x_1)$.

```
1. $\Gamma := \{\mathsf{G} \Rightarrow Q[\mathbf{k}, \mathsf{e}]\}$ where $\mathbf{k}$ consists of distinct fresh constants
2. Repeat
       If there is a model $\mathcal{I}$ of $T$ satisfying $\Gamma$ and $\mathsf{G}$
       then let $\Gamma := \Gamma \cup \{\neg Q[\mathbf{k}, t[\mathbf{k}]]\}$ for some $\Sigma$-term $t[\boldsymbol{x}]$ such that $t[\mathbf{k}]^{\mathcal{I}} = \mathsf{e}^{\mathcal{I}}$;
       otherwise, return "no solution found"
   until $\Gamma$ contains a $T$-unsatisfiable set $\{\neg Q[\mathbf{k}, t_1[\mathbf{k}]], \ldots, \neg Q[\mathbf{k}, t_p[\mathbf{k}]]\}$
3. Return $\lambda \boldsymbol{x}.\, \mathsf{ite}(Q[\boldsymbol{x}, t_p[\boldsymbol{x}]], t_p[\boldsymbol{x}],\ (\cdots \mathsf{ite}(Q[\boldsymbol{x}, t_2[\boldsymbol{x}]], t_2[\boldsymbol{x}], t_1[\boldsymbol{x}]) \cdots))$ for $f$
```

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

Example 1. Let T be the theory of linear integer arithmetic with the usual signature and integer sort Int. Let $\boldsymbol{x} = (x_1, x_2)$. Now consider the property

$$P[f, \boldsymbol{x}] := f(\boldsymbol{x}) \geq x_1 \wedge f(\boldsymbol{x}) \geq x_2 \wedge (f(\boldsymbol{x}) \approx x_1 \vee f(\boldsymbol{x}) \approx x_2) \tag{5}$$

with f of type $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ and x_1, x_2 of type Int. The synthesis problem $\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ is solved exactly by the function that returns the maximum of its two inputs. Since P is a single-invocation property, we can solve that problem by proving the T-unsatisfiability of the conjecture $\exists \boldsymbol{x}\, \forall y\, \neg Q[\boldsymbol{x}, y]$ where

$$Q[\boldsymbol{x}, y] := y \geq x_1 \wedge y \geq x_2 \wedge (y \approx x_1 \vee y \approx x_2) \tag{6}$$

After Skolemization the conjecture becomes $\forall y\, \neg Q[\mathbf{a}, y]$ for fresh constants $\mathbf{a} = (\mathsf{a}_1, \mathsf{a}_2)$. When asked to determine the satisfiability of that conjecture an SMT solver may, for instance, instantiate it with a_1 and then a_2 for y, producing the T-unsatisfiable set $\{\neg Q[\mathbf{a}, \mathsf{a}_1], \neg Q[\mathbf{a}, \mathsf{a}_2]\}$. By Proposition 1, one solution for $\forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ is $f = \lambda \boldsymbol{x}.\, \mathsf{ite}(Q[\boldsymbol{x}, x_2], x_2, x_1)$, which simplifies to $\lambda \boldsymbol{x}.\, \mathsf{ite}(x_2 \geq x_1, x_2, x_1)$, representing the desired maximum 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 $\forall y\, \neg Q[\mathbf{k}, y]$. Typically, SMT solvers that reason about quantified formulas use heuristic quantifier instantiation techniques based on E-matching [9], which instantiates universal quantifiers with terms occurring in some current set of ground terms built incrementally from the input formula. Using 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 (4).

The new technique is similar to a popular scheme for synthesis known as counterexample-guided inductive synthesis, implemented in various synthesis approaches (e.g., [14, 29]), 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 $\neg Q[\mathbf{k}, y]$ starting with the formula $\mathsf{G} \Rightarrow Q[\mathbf{k}, \mathsf{e}]$ 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 Γ is still satisfiable but all of its models falsify G, in which case the search for a solution was inconclusive. The procedure is not *solution-complete*, that is, it is not 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 Step 2 of the procedure is intentionally left underspecified because it can be done in a number of ways. Having a good heuristic for such instantiations is, however, critical to the effectiveness of the procedure in practice. In a Σ-theory T, like integer arithmetic, with a fixed interpretation for symbols in Σ and a distinguished set of ground Σ-terms denoting the elements of a sort, a simple, if naive, choice for t in Figure 1 is the distinguished term denoting the element $\mathsf{e}^{\mathcal{I}}$. For instance, if σ is Int in integer 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. For instance, 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 (6) 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 $Q[\mathbf{a}, \mathsf{e}]$ with $\mathbf{a} = (\mathsf{a}_1, \mathsf{a}_2)$ only if it can first find a subset M of that formula's literals that collectively entail $Q[\mathbf{a}, \mathsf{e}]$ at the propositional level. Due to the last conjunct of (6), M must include either $\mathsf{e} \approx \mathsf{a}_1$ or $\mathsf{e} \approx \mathsf{a}_2$. Hence, whenever a model can be constructed for $Q[\mathbf{a}, e]$, the equivalence class containing e must contain either a_1 or a_2. 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 instances $\neg Q[\mathbf{a}, \mathsf{a}_1]$ and $\neg Q[\mathbf{a}, \mathsf{a}_2]$ to Γ. As noted in Example 1, these two instances are jointly T-unsatisfiable. We expect that more sophisticated instantiation techniques can be incorporated. In particular, both quantifier elimination techniques [7, 21] and approaches currently used to infer invariants from templates [8, 19] 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, 22] these restrictions are expressed by a grammar R (augmented with a kind of *let* binder) defining the language of solution terms, or *programs*, for the synthesis problem. In this section, we present a variant of the approach in the previous section that incorporates the syntactic restriction directly into the SMT solver via a

$$\begin{array}{ll}
\forall x\, y\; \mathsf{ev}(\mathsf{x}_1, x, y) \approx x & \forall s_1\, s_2\, x\, y\; \mathsf{ev}(\mathsf{leq}(s_1, s_2), x, y) \approx (\mathsf{ev}(s_1, x, y) \leq \mathsf{ev}(s_2, x, y)) \\
\forall x\, y\; \mathsf{ev}(\mathsf{x}_2, x, y) \approx y & \forall s_1\, s_2\, x\, y\; \mathsf{ev}(\mathsf{eq}(s_1, s_2), x, y) \approx (\mathsf{ev}(s_1, x, y) \approx \mathsf{ev}(s_2, x, y)) \\
\forall x\, y\; \mathsf{ev}(\mathsf{zero}, x, y) \approx 0 & \forall c_1\, c_2\, x\, y\; \mathsf{ev}(\mathsf{and}(c_1, c_2), x, y) \approx (\mathsf{ev}(c_1, x, y) \wedge \mathsf{ev}(c_2, x, y)) \\
\forall x\, y\; \mathsf{ev}(\mathsf{one}, x, y) \approx 1 & \forall c\, x\, y\; \mathsf{ev}(\mathsf{not}(c), x, y) \approx \neg \mathsf{ev}(c, x, y) \\
\multicolumn{2}{l}{\forall s_1\, s_2\, x\, y\; \mathsf{ev}(\mathsf{plus}(s_1, s_2), x, y) \approx \mathsf{ev}(s_1, x, y) + \mathsf{ev}(s_2, x, y)} \\
\multicolumn{2}{l}{\forall s_1\, s_2\, x\, y\; \mathsf{ev}(\mathsf{minus}(s_1, s_2), x, y) \approx \mathsf{ev}(s_1, x, y) - \mathsf{ev}(s_2, x, y)} \\
\multicolumn{2}{l}{\forall c\, s_1\, s_2\, x\, y\; \mathsf{ev}(\mathsf{if}(c, s_1, s_2), x, y) \approx \mathsf{ite}(\mathsf{ev}(c, x, y), \mathsf{ev}(s_1, x, y), \mathsf{ev}(s_2, x, y))}
\end{array}$$

Fig. 2. Axiomatization of the evaluation operators in grammar R from Example 2.

deep embedding of the syntactic restriction R into the solver's logic. The main idea is to represent R as a set of algebraic datatypes and build into the solver an interpretation of these datatypes 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 here with a concrete example.

Example 2. Consider again the synthesis conjecture (6) from Example 1 but now with a syntactic restriction R for the solution space expressed by these algebraic datatypes:

$$\begin{array}{lcl}
\mathsf{S} & := & \mathsf{x}_1 \mid \mathsf{x}_2 \mid \mathsf{zero} \mid \mathsf{one} \mid \mathsf{plus}(\mathsf{S}, \mathsf{S}) \mid \mathsf{minus}(\mathsf{S}, \mathsf{S}) \mid \mathsf{if}(\mathsf{C}, \mathsf{S}, \mathsf{S}) \\
\mathsf{C} & := & \mathsf{leq}(\mathsf{S}, \mathsf{S}) \mid \mathsf{eq}(\mathsf{S}, \mathsf{S}) \mid \mathsf{and}(\mathsf{C}, \mathsf{C}) \mid \mathsf{not}(\mathsf{C})
\end{array}$$

The datatypes are meant to encode a term signature that includes nullary constructors for the variables x_1 and x_2 of (6), 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 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_1 and x_2.

It is possible to instrument an SMT solver that support 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 $\mathsf{ev}(d, t_1, t_2)$ to smaller terms by means of selected instantiations of the axioms from Figure 2, with a number of instances proportional to the size of term d. It is also possible to show that T_{D} is satisfaction complete with respect to the class

$$\mathbf{L}_2 := \{\exists g\, \forall \boldsymbol{z}\, P[\lambda \boldsymbol{z}.\, \mathsf{ev}(g, \boldsymbol{z}),\, \boldsymbol{x}] \mid P[f, \boldsymbol{x}] \in \mathbf{P}\}$$

where instead of terms of the form $f(t_1, t_2)$ in P we have, modulo β-reductions, terms of the form $\mathsf{ev}(g, t_1, t_2)$.[6] For instance, the formula $P[f, \boldsymbol{x}]$ in Equation (5) from Exam-

[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.

```
1. $\Gamma := \emptyset$
2. Repeat
   (a) Let $\mathsf{k}$ be a tuple of distinct fresh constants.
       If there is a model $\mathcal{I}$ of $T_{\mathrm{D}}$ satisfying $\Gamma$ *and* $\mathsf{G}$, then $\Gamma := \Gamma \cup \{\neg P_{\mathsf{ev}}[\mathsf{e}^{\mathcal{I}}, \mathsf{k}]\}$ ;
       otherwise, return "no solution found"
   (b) If there is a model $\mathcal{J}$ of $T_{\mathrm{D}}$ satisfying $\Gamma$, then $\Gamma := \Gamma \cup \{\mathsf{G} \Rightarrow P_{\mathsf{ev}}[\mathsf{e}, \mathsf{k}^{\mathcal{J}}]\}$ ;
       otherwise, return $\mathsf{e}^{\mathcal{I}}$ as a solution
```

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

ple 1 can be restated in T_{D} as the formula below where g is a variable of type S:

$$P_{\mathsf{ev}}[g, \boldsymbol{x}] := \mathsf{ev}(g, \boldsymbol{x}) \geq x_1 \wedge \mathsf{ev}(g, \boldsymbol{x}) \geq x_2 \wedge (\mathsf{ev}(g, \boldsymbol{x}) \approx x_1 \vee \mathsf{ev}(g, \boldsymbol{x}) \approx x_2)$$

In contrast to $P[f, \boldsymbol{x}]$, the new formula $P_{\mathsf{ev}}[g, \boldsymbol{x}]$ is first-order, with the role of the second-order variable f now played by the first-order variable g.

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 $\forall g\, \exists \boldsymbol{x}\, \neg P_{\mathsf{ev}}[g, \boldsymbol{x}]$. Instantiating g in the latter formula with $s := \mathsf{if}(\mathsf{leq}(\mathsf{x}_1, \mathsf{x}_2), \mathsf{x}_2, \mathsf{x}_1)$, say, produces a formula that the solver can prove to be T_{D}-unsatisfiable. This suffices to show that the program $\mathsf{ite}(x_1 \leq x_2, x_2, x_1)$, the analogue of s in the language of T, is a solution of the synthesis conjecture (5) under the syntactic restriction R. ■

To prove the unsatisfiability of formulas like $\forall g\, \exists \boldsymbol{x}\, \neg P_{\mathsf{ev}}[g, \boldsymbol{x}]$ in the example above 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 representing the restricted solution space. 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 in Step 2b the value of e in the model $\mathcal{I}$ found in Step 2a. As we showed in the previous example, a program that solves the original problem can then be reconstructed from the returned datatype term.

Implementation. We implemented the procedure in the CVC4 solver. 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 2a, the procedure finds the candidate $\mathsf{if}(\mathsf{leq}(\mathsf{x}_1, \mathsf{x}_2), \mathsf{x}_2, \mathsf{x}_1)$, 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.

To make the procedure practical it is necessary to look for *small* solutions to synthesis conjectures. A simple way to limit the size of the candidate solutions is to consider smaller programs before larger ones. Adapting techniques for finding finite models of minimal size [26], we use a strategy that starting, from $n = 0$, searches for programs of size $n + 1$ only after its has exhausted the search for programs of size n. In solvers based on the DPLL(T) architecture, like CVC4, 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

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

For $i = 1, \ldots, 6$, a_i and b_i are fresh constants of type Int.

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

datatype sort σ and stands 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 extended the procedure for algebraic 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 terms of the form $\mathsf{size}(u)$, for each variable or uninterpreted constant u of datatype sort in the problem. This is enough for our purposes since the only term u like that in our synthesis procedure is e.

Proposition 2. With the search 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.

Note that by this proposition the procedure can diverge only if the input synthesis conjecture has no solution. We refer the reader to a longer version of this paper for a proof of Proposition 2 [23]. For a general idea, the proof of solution soundness is based on the observation that when the procedure terminates at Step 2b, Γ has an unsatisfiable core with just one instance of $\neg P[g, \boldsymbol{x}]$. The procedure is refutation sound since when no model of Γ in Step 2a satisfies G, we have that even an arbitrary e cannot satisfy the current set of instances added to Γ in Step 2b. Finally, the procedure is solution complete first of all because Step 2a and 2b are effective thanks to the decidability of the background theory T_{D}. Each execution of Step 2a is guaranteed to produce a new candidate since T_{D} is also satisfaction complete. Thus, in the worst case, the procedure amounts an enumeration of all possible programs until a solution is found.

5 Single Invocation Techniques for Syntax-Guided Problems

In this section, we considered the combined case of *single-invocation synthesis conjectures with syntactic restrictions*. Given a set R of syntactic restrictions expressed by a datatype S for programs and a datatype C for Boolean expressions, consider the case where (i) S contains the constructor $\mathsf{if} : \mathsf{C} \times \mathsf{S} \times \mathsf{S} \to \mathsf{S}$ (with the expected meaning) and (ii) the function to be synthesized is specified by a single-invocation property that can be expressed as a term of sort C. This is the case for the conjecture from Example 2 where the property $P_{\mathsf{ev}}[g, \boldsymbol{x}]$ can be rephrased as:

$$P_{\mathsf{C}}[g, \boldsymbol{x}] := \mathsf{ev}(\mathsf{and}(\mathsf{leq}(\mathsf{x}_1, g), \mathsf{and}(\mathsf{leq}(\mathsf{x}_2, g), \mathsf{or}(\mathsf{eq}(g, \mathsf{x}_1), \mathsf{eq}(g, \mathsf{x}_2)))), \boldsymbol{x}) \quad (7)$$

where again g has type S, $\boldsymbol{x} = (x_1, x_2)$, and x_1 and x_2 have type Int. The procedure in Figure 1 can be readily modified to apply to this formula, with $P_{\mathsf{C}}[g, \mathbf{k}]$ and g taking the role respectively of $Q[\mathbf{k}, y]$ and y in that figure, since it generates solutions meeting our syntactic requirements. Running this modified procedure instead the one in Figure 3 has the advantage that only the outputs of a solution need to be synthesized, not conditions in ite-terms. However, in our experimental evaluation found that the overhead of using an embedding into datatypes for syntax-guided problems is significant with respect to the performance of the solver on problems with no syntactic restrictions. For this reason, we advocate an approach for single-invocation synthesis conjectures with syntactic restrictions that runs the procedure from Figure 1 as is, ignoring the syntactic restrictions R, and subsequently reconstructs from its returned solution one satisfying the restrictions. For that it is useful to assume that terms t in T can be effectively reduced to some (T-equivalent and unique) *normal form*, which we denote by $t\!\downarrow$.

Say the procedure from Figure 1 returns a solution $\lambda \boldsymbol{x}.\, t$ for a function f. To construct from that a solution that meets the syntactic restrictions specified by datatype S, we run the iterative procedure described in Figure 5. This procedure maintains an evolving set A of triples of the form (t, s, D), where D is a datatype, t is a term in normal form, s is a term satisfying the restrictions specified by D. The procedure incrementally makes calls to the subprocedure rcon, which takes a normal form term t, a datatype D and the set A above, and returns a pair (s, U) where s is a term equivalent to t in T, and U is a set of pairs (s', D') where s' is a subterm of s that fails to satisfy the syntactic restriction expressed by datatype D'. Overall, the procedure alternates between calling rcon and adding triples to A until $\mathrm{rcon}(t, D, A)$ returns a pair of the form $(s, \emptyset)$, in which case s is a solution satisfying the syntactic restrictions specified by S.

Example 3. Say we wish to construct a solution equivalent to $\lambda x_1\, x_2.\, x_1 + (2 * x_2)$ that meets restrictions specified by datatype S from Example 2. To do so, we let $A = \emptyset$, and call $\mathrm{rcon}((x_1 + (2 * x_2))\downarrow, \mathsf{S}, A)$. Since A is empty and $+$ is the analogue of constructor $\mathsf{plus}^{\mathsf{SSS}}$ of S, assuming $(x_1 + (2 * x_2))\!\downarrow = x_1 + (2 * x_2)$, we may choose to return a pair based on the result of calling rcon on $x_1\!\downarrow$ and $(2 * x_2)\!\downarrow$. Since $\mathsf{x}_1^{\mathsf{S}}$ is a constructor of S and $x_1\!\downarrow = x_1$, $\mathrm{rcon}(x_1, \mathsf{S}, A)$ returns $(x_1, \emptyset)$. Since S does not have a constructor for $*$, we must either choose a term t such that $t\!\downarrow = (2 * x_2)\!\downarrow$ where the topmost symbol of t is the analogue of a constructor in S, or otherwise return the pair $(2 * x_2, \{(2 * x_2, \mathsf{S})\})$. Suppose we do the latter, and thus $\mathrm{rcon}(x_1 + (2 * x_2), \mathsf{S}, A)$ returns $(x_1 + (2 * x_2), \{(2 * x_2, \mathsf{S})\})$. Since the second component of this pair is not

```
1. $A := \emptyset$ ; $t' := t\downarrow$
2. for $i = 1, 2, \ldots$
   (a) $(s, U) := \mathrm{rcon}(t', \mathsf{S}, A)$;
   (b) if $U$ is empty, return $s$; otherwise, for each datatype $D_j$ occurring in $U$
           let $d_i$ be the $i^{th}$ term in a fair enumeration of the elements of $D_j$
           let $t_i$ be the analogue of $d_i$ in the background theory $T$
           add $(t_i\downarrow, t_i, D_j)$ to $A$

rcon$(t, D, A)$
   if $(t, s, D) \in A$, return $(s, \emptyset)$; otherwise, do one of the following:
   (1) choose a $f(t_1, \ldots, t_n)$ s.t. $f(t_1, \ldots, t_n)\downarrow = t$ and $f$ has an analogue $c^{D_1 \ldots D_n D}$ in $D$
       let $(s_i, U_i) = \mathrm{rcon}(t_i\downarrow, D_i, A)$ for $i = 1, \ldots, n$
       return $(f(s_1, \ldots, s_n), U_1 \cup \ldots \cup U_n)$
   (2) return $(t, \{(t, D)\})$
```

Fig. 5. A procedure for finding a term equivalent to t that meets the syntactic restrictions specified by datatype S.

empty, we pick in Step 2b the first element of S, x_1 say, and add (x_1, x_1, S) to A. We then call $\mathrm{rcon}((x_1 + (2 * x_2))\downarrow, \mathsf{S}, A)$ which by the same strategy above returns $(x_1 + (2*x_2), \{(2*x_2, \mathsf{S})\})$. This process continues until we pick, the term $\mathsf{plus}(\mathsf{x}_2, \mathsf{x}_2)$ say, whose analogue is $x_2 + x_2$. Assuming $(x_2+x_2)\downarrow = (2*x_2)\downarrow$, after adding the pair $(2*x_2, x_2+x_2, \mathsf{S})$ to A, $\mathrm{rcon}((x_1+(2*x_2))\downarrow, \mathsf{S}, A)$ returns the pair $(x_1+(x_2+x_2), \emptyset)$, indicating that $\lambda x_1\, x_2.\, x_1 + (x_2 + x_2)$ is equivalent to $\lambda x_1\, x_2.\, x_1 + (2 * x_2)$, and meets the restrictions specified by S. ■

This procedure depends upon the use of normal forms for terms. It should be noted that, since the top symbol of t is generally ite, this normalization includes both low-level rewriting of literals within t, but also includes high-level rewriting techniques such as ite simplification, redundant subterm elimination and destructive equality resolution. Also, notice that we are not assuming that $t\downarrow = s\downarrow$ if and only if t is equivalent to s, and thus normal forms only underapproximate an equivalence relation between terms. Having a (more) consistent normal form for terms allows us to compute a (tighter) underapproximation, thus improving the performance of the reconstruction. In this procedure, we use the same normal form for terms that is used by the individual decision procedures of CVC4. This is unproblematic for theories such as linear arithmetic whose normal form for terms is a sorted list of monomials, but it can be problematic for theories such as bitvectors. As a consequence, we use several optimizations, omitted in the description of the procedure in Figure 5, to increase the likelihood that the procedure terminates in a reasonable amount of time. For instance, in our implementation the return value of rcon is not recomputed every time A is updated. Instead, we maintain an evolving directed acyclic graph (dag), whose nodes are pairs (t, S) for term t and datatype S (the terms we have yet to reconstruct), and whose edges are the direct subchildren of that term. Datatype terms are enumerated for all datatypes in this dag, which is incrementally pruned as pairs are added to A until it becomes empty. Another optimization is that the procedure rcon may choose to try simultaneously to reconstruct *multiple* terms of the form $f(t_1, \ldots, t_n)$ when matching a term t to a syntactic specification S, reconstructing t when any such term can be reconstructed.

	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	4	2250.7	2	71.2	50	878.5	0	0	5	1416.7	2	0.0	7	0.6	70	4617.7
cvc4+sg	1	3.1	0	0	34	4308.9	1	0.5	3	1.7	2	0.5	7	628.3	48	4943
cvc4+si-r	(32)	1.2	(6)	4.7	(56)	2.1	(43)	3403.5	(15)	0.6	(8)	1.0	(8)	0.2	(168)	3413.3
cvc4+si	30	1449.5	5	0.1	52	2322.9	0	0	6	0.1	2	0.5	7	0.1	102	3773.2

Fig. 6. Results for single-invocation synthesis conjectures, showing times (in seconds) and number of benchmarks solved by each solver and configuration over 8 benchmark classes with a 3600s timeout. The number of benchmarks solved by configuration **cvc4+si-r** are in parentheses because its solutions do not necessarily satisfy the given syntactic restrictions.

Although the overhead of this procedure can be significant when large subterms do not meet the syntactic restrictions, we found that in practice it quickly terminates successfully for a majority of the solutions we considered where reconstruction was possible, as we discuss in the next section. Furthermore, it makes our implementation more robust, since it effectively treats in the same way different properties that are equal modulo normalization (which is parametric in the built-in theories we consider).

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 algebraic datatypes. We evaluated our implementation on 243 benchmarks used in the SyGuS 2014 competition [1] that were publicly available on the StarExec execution service [31]. The benchmarks are in a new format for specifying syntax-guided synthesis problems [22]. We added parsing support to CVC4 for most features of this format. All SyGuS benchmarks considered contain synthesis conjectures whose background theory is either linear integer arithmetic or bitvectors. We made some minor modifications to benchmarks to avoid naming conflicts, and to explicitly define several bitvector operators that are not supported natively by CVC4.

We considered multiple configurations of CVC4 corresponding to the techniques mentioned in this paper. Configuration **cvc4+sg** executes the syntax-guided procedure from Section 4, even in cases where the synthesis conjecture is single-invocation. Configuration **cvc4+si-r** executes the procedure from Section 3 on all benchmarks having conjectures that it can deduce are single-invocation. In total, it discovered that 176 of the 243 benchmarks could be rewritten into a form that was single-invocation. This configuration simply ignores any syntax restrictions on the expected solution. Finally, configuration **cvc4+si** uses the same procedure used by **cvc4+si-r** but then attempts to reconstruct any found solution as a term in required syntax, as described in Section 5.

We ran all configurations on all benchmarks on the StarExec cluster.[7] We provide comparative results here primarily against the enumerative CEGIS solver ESOLVER [32], the winner of the SyGuS 2014 competition. In our tests, we found that ESOLVER performed significantly better than the other entrants of that competition.

Benchmarks with single-invocation synthesis conjectures. The results for benchmarks with single-invocation properties are shown in Figure 6. Configuration **cvc4+si-r**

[7] A detailed summary can be found at `http://lara.epfl.ch/w/cvc4-synthesis`.

	int (3)		**invgu** (28)		**invg** (28)		**vctrl** (8)		**Total** (67)	
	#	time	#	time	#	time	#	time	#	time
esolver	3	1.6	25	86.3	25	85.6	5	29.5	58	203.0
cvc4+sg	3	1476.0	23	811.6	22	2283.2	5	2933.1	53	7503.9

Fig. 7. Results for synthesis conjectures that are not single-invocation, showing times (in seconds) and numbers of benchmarks solved by CVC4 and ESOLVER over 4 benchmark classes with a 3600s timeout.

found a solution (although not necessarily in the required language) very quickly for a majority of benchmarks. It terminated successfully for 168 of 176 benchmarks, and in less than a second for 159 of those. Not all solutions found using this method met the syntactic restrictions. Nevertheless, our methods for reconstructing these solutions into the required grammar, implemented in configuration **cvc4+si**, succeeded in 102 cases, or 61% of the total. This is 32 more benchmarks than the 70 solved by ESOLVER, the best known solver for these benchmarks so far. In total, **cvc4+si** solved 34 benchmarks that ESOLVER did not, while ESOLVER solved 2 that **cvc4+si** did not.

The solutions returned by **cvc4+si-r** were often large, having an order of 10K subterms for harder benchmarks. However, after exhaustively applying simplification techniques during reconstruction with configuration **cvc4+si**, we found that the size of those solutions is comparable to other solvers, and in some cases even smaller. For instance, among the 68 benchmarks solved by both ESOLVER and **cvc4+si**, the former produced a smaller solution in 15 cases and the latter in 9. Only in 2 cases did **cvc4+si** produce a solution that had 10 more subterms than the solution produced by ESOLVER. This indicates that in addition to having a high precision, the techniques from Section 5 used for solution reconstruction are effective also at producing succinct solutions for this benchmark library.

Configuration **cvc4+sg** does not take advantage of the fact that a synthesis conjecture is single-invocation. However, it was able to solve 48 of these benchmarks, including a small number not solved by any other configuration, like one from the **icfp** class whose solution was a single argument function over bitvectors that shifted its input right by four bits. In addition to being solution complete, **cvc4+sg** always produces solutions of minimal term size, something not guaranteed by the other solvers and CVC4 configurations. Of the 47 benchmarks solved by both **cvc4+sg** and ESOLVER, the solution returned by **cvc4+sg** was smaller than the one returned by ESOLVER in 6 cases, and had the same size in the others. This provides an experimental confirmation that the fairness techniques for term size described in Section 4 ensure minimal size solutions.

Benchmarks with non-single-invocation synthesis conjectures. Configuration **cvc4+sg** is the only CVC4 configuration that can process benchmarks with synthesis conjectures that are not single-invocation. The results for ESOLVER and **cvc4+sg** on such benchmarks from SyGuS 2014 are shown in Figure 7. Configuration **cvc4+sg** solved 53 of them over a total of 67. ESOLVER solved 58 and additionally reported that 6 had no solution. In more detail, ESOLVER solved 7 benchmarks that **cvc4+sg** did not, while **cvc4+sg** solved 2 benchmarks (from the **vctrl** class) that ESOLVER could not solve. In terms of precision, **cvc4+sg** is quite competitive with the state of the art on these benchmarks. To give other points of comparison, at the SyGuS 2014 competition [1] the second best solver (the Stochastic solver) solved 40 of these benchmarks within a one hour limit and Sketch solved 23.

n	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. 8. 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.

Overall results. In total, over the entire SyGuS 2014 benchmark set, 155 benchmarks can be solved by a configuration of CVC4 that, whenever possible, runs the methods for single-invocation properties described in Section 3, and otherwise runs the method described in Section 4. This number is 27 higher than the 128 benchmarks solved in total by ESOLVER. Running both configuration **cvc4+sg** and **cvc4+si** in parallel[8] solves 156 benchmarks, indicating that CVC4 is highly competitive with state-of-the-art tools for syntax guided synthesis. CVC4's performance is noticeably better than ESOLVER on single-invocation properties, where our new quantifier instantiation techniques give it a distinct advantage.

Competitive advantage on single-invocation properties in the presence of ite. We conclude by observing that for certain classes of benchmarks, configuration **cvc4+si** scales significantly better than state-of-the-art synthesis tools. Figure 8 shows this in comparison with ESOLVER for the problem of synthesizing a function that computes the maximum of n integer inputs. As reported by Alur et al. [1], no solver in the SyGuS 2014 competition was able to synthesize such a function for $n = 5$ within one hour.

For benchmarks from the **array** class, whose 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.[9] In contrast, **cvc4+si** was able to reconstruct solutions for arrays of size 15 (the largest benchmark in the class) in 0.3 seconds, and solved each of the benchmarks in the class but 8 within 1 second.

7 Conclusion

We have shown that SMT solvers, instead of just acting as subroutines for automated software synthesis tasks, can be instrumented to perform synthesis themselves. We have presented a few approaches for enabling SMT solvers to construct solutions for the broad class of syntax-guided synthesis problems and discussed their implementation in CVC4. This is, to the best of our knowledge, the first implementation of synthesis inside an SMT solver and it already shows considerable promise. Using a novel quantifier instantiation technique and a solution enumeration technique for the theory of algebraic datatypes, our implementation is competitive with the state of the art represented by the systems that participated in the 2014 syntax-guided synthesis competition. Moreover, for the important class of single-invocation problems when syntax restrictions permit the if-then-else operator, our implementation significantly outperforms those systems.

Acknowledgments. We would like to thank Liana Hadarean for helpful discussions on the normal form used in CVC4 for bit vector terms.

[8] CVC4 has a *portfolio* mode that allows it to run multiple configurations at the same time.

[9] These benchmarks, as contributed to the SyGuS benchmark set, use integer variables only; they were generated by expanding fixed-size arrays and contain no operations on 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 J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *LNCS*, pages 316–330. Springer, 2010.
8. 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.
9. 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.
10. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *CACM*, 55(2):103–111, 2012.
18. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Functional synthesis for linear arithmetic and sets. *STTT*, 15(5-6):455–474, 2013.

19. R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In A. Biere and R. Bloem, editors, *CAV*, volume 8559 of *LNCS*, pages 762–778. Springer, 2014.
20. Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *TOPLAS*, 2(1):90–121, 1980.
21. 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.
22. M. Raghothaman and A. Udupa. Language to specify syntax-guided synthesis problems. *CoRR*, abs/1405.5590, 2014.
23. A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. On counterexample guided quantifier instantiation for synthesis in CVC4. *CoRR*, abs/1502.04464, 2015. `http://arxiv.org/abs/1502.04464`.
24. 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.
25. 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.
26. A. J. Reynolds. *Finite Model Finding in Satisfiability Modulo Theories*. PhD thesis, The University of Iowa, 2013.
27. 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.
28. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
29. 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.
30. S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.
31. A. Stump, G. Sutcliffe, and C. Tinelli. Starexec: a cross-community infrastructure for logic solving. In *Proceedings of the 7th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2014.
32. A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. In *PLDI*, pages 287–296. ACM, 2013.