Form Methods Syst Des
DOI 10.1007/s10703-017-0270-2

Refutation-based synthesis in SMT

Andrew Reynolds[1,2] · **Viktor Kuncak**[1] · **Cesare Tinelli**[2] ·
Clark Barrett[3] · **Morgan Deters**[4]

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.

Keywords Program synthesis · Satisfiability modulo theories · Automated deduction

In memory of Morgan Deters who passed away unexpectedly in 2015.

✉ Cesare Tinelli
cesare-tinelli@uiowa.edu

Andrew Reynolds
andrew-reynolds@uiowa.edu

Viktor Kuncak
viktor.kuncak@epfl.ch

Clark Barrett
barrett@cs.stanford.edu

1 École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland

2 Department of Computer Science, The University of Iowa, Iowa City, IA, USA

3 Department of Computer Science, Stanford University, Stanford, CA, USA

4 Department of Computer Science, New York University, New York, NY, USA

Published online: 16 February 2017

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 [27,28] but also translates to imperative programs through techniques that include bounding input space, verification condition generation, and invariant discovery [45–47]. Function synthesis is also an important subtask in the synthesis of protocols and reactive systems, especially when these systems are infinite-state [4,42]. The SyGuS format and competition [2,3,37], inspired by the success of the SMT-LIB and SMT-COMP efforts [6], has significantly improved and simplified the process of rigorously comparing different solvers on synthesis problems.

The connection between synthesis and theorem proving was established already in early work on the subject [18,30]. It is notable that early research [30] 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 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 [24] 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.

In this paper, which is an extended and improved version of [38], 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 the CVC4 SMT solver significantly outperforms leading tools from the SyGuS competition.

1.1 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.

Signatures We fix a large enough 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 denote their union by $\mathbf{X}$. 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*. Note that we consider only *first-order* function symbols, that is, function symbols whose input and output types are all first-order types. We drop the sort superscript from variables or function symbols when it is clear from context or unimportant. We assume that, unless stated otherwise, signatures always include a Boolean sort Bool and constants tt and ff of type Bool (respectively, for true and false). The *union* $\Sigma_1 \cup \Sigma_2$ of a signature Σ_1 and a signature Σ_2 is the signature Σ such that $\Sigma^{\mathrm{s}} = \Sigma_1^{\mathrm{s}} \cup \Sigma^{\mathrm{s}}$ and $\Sigma^{\mathrm{f}} = \Sigma_1^{\mathrm{f}} \cup \Sigma^{\mathrm{f}}$. A signature Σ_1 is a *subsignature* of a signature Σ if $\Sigma = \Sigma_1 \cup \Sigma_2$ for some signature Σ_2.

Term and formulas Given a many-sorted signature Σ together with quantifiers and lambda abstractions $\lambda x_1^{\sigma_1} \ldots x_n^{\sigma_n}\, t$, 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. Note that 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, often write just t in place of the atom $t \approx \mathsf{tt}$, to simplify the notation. A Σ-term/formula is *ground* if it has no variables, and it is *first-order* if it has only *first-order variables*, that is, variables of first-order type. Free and bound occurrences of a variable in a formula are also defined as usual. A (Σ-)*sentence* is a (Σ-)formula with no free variables.

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 s is a Σ-term or formula and $\boldsymbol{x} = (x_1, \ldots, x_n)$ has no repeated variables, we write $s[\boldsymbol{x}]$ to denote that all of s's free variables are from $\boldsymbol{x}$; if $\boldsymbol{t} = (t_1, \ldots, t_n)$ is a term tuple, we write $s[\boldsymbol{t}]$ for the term or formula obtained from s by simultaneously replacing, for all $i = 1, \ldots, n$, every occurrence of x_i in s by t_i. When convenient, we will treat a tuple like $\boldsymbol{x}$ or $\boldsymbol{t}$ as the set of its elements.

Interpretations 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 $\mathsf{Bool}^{\mathcal{I}} = \{\mathsf{tt}, \mathsf{ff}\}$, $\mathsf{ff}^{\mathcal{I}} = \mathsf{ff}$ and $\mathsf{tt}^{\mathcal{I}} = \mathsf{tt}$; 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 $n = 0$. If $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ are distinct variables and $e_1, \ldots, e_n$ are domain elements with $e_1 \in \sigma_1^{\mathcal{I}}, \ldots, e_n \in \sigma_n^{\mathcal{I}}$, we denote by $\mathcal{I}[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ the Σ-interpretation that maps each x_i to e_i and is otherwise identical to $\mathcal{I}$. The interpretation $\mathcal{I}$ induces a mapping from terms t of sort σ to elements $t^{\mathcal{I}}$ of $\sigma^{\mathcal{I}}$ as expected. A satisfiability relation between Σ-interpretations and Σ-formulas or sets thereof is defined inductively as usual. A satisfying interpretation for a Σ-formula φ *models (or is a model of)* φ. A Σ-interpretation $\mathcal{I}$ is *term-generated* if each of its domain elements is denoted by a ground Σ-term, that is, if for all $\sigma \in \Sigma^{\mathrm{s}}$ and all $e \in \sigma^{\mathcal{I}}$ there is a ground Σ-term t of sort σ such that $e = t^{\mathcal{I}}$.

If Ω is a subsignature of a signature Σ, the Ω-*reduct* of a Σ-interpretation $\mathcal{I}$ is an Ω-interpretation that interprets its sort and function symbols exactly as $\mathcal{I}$.

An *isomorphism* from a Σ-interpretation $\mathcal{I}$ to a Σ-interpretation $\mathcal{J}$ is a family of bijective mappings $\{h_\sigma : \sigma^{\mathcal{I}} \to \sigma^{\mathcal{J}} \mid \sigma \in \Sigma^{\mathrm{s}}\}$ such that for every $f \in \Sigma^{\mathrm{f}}$ of type $\sigma_1 \times \cdots \times \sigma_n \to \sigma_{n+1}$, and every $(v_1, \ldots, v_n) \in \sigma_1^{\mathcal{I}} \times \cdots \times \sigma_n^{\mathcal{I}}$,

$$h_{\sigma_{n+1}}(f^{\mathcal{I}}(v_1, \ldots, v_n)) = f^{\mathcal{J}}(h_{\sigma_1}(v_1), \ldots, h_{\sigma_n}(v_n))$$

Two Σ-interpretations $\mathcal{I}$ and $\mathcal{J}$ are *isomorphic*, written $\mathcal{I} \equiv \mathcal{J}$, if there is an isomorphism from one to the other. Among other things, it is possible to show that $\equiv$ is an equivalence relation and that two isomorphic interpretations satisfy exactly the same Σ-sentences.

Theories 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 isomorphism.[1] A Σ-formula φ is T-*satisfiable* (resp., T-*unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A formula φ is T-*valid*, written $\models_T \varphi$, if every model of T is a model of φ. A set Γ of formulas T-*entails* a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. Two Σ-formulas are T-*equivalent* if they T-entail each other. Two Σ-terms s and t are T-*equivalent* if $s \approx t$ is T-valid.

If T_1 is a Σ_1-theory and T_2 is a Σ_2-theory, the *union* $T_1 \cup T_2$ of T_1 and T_2, when it exists, is the $(\Sigma_1 \cup \Sigma_2)$-theory whose set of models consists of all the $(\Sigma_1 \cup \Sigma_2)$-interpretations whose Σ_i-reduct is a model of T_i for $i = 1, 2$.[2]

The *theory of a* Σ-*structure* $\mathcal{I}$ is the theory $T = (\Sigma, \mathbf{I})$ where $\mathbf{I}$ consists of all the Σ-structures that are isomorphic to $\mathcal{I}$.

Sometimes we will extend the signature Σ of a theory T with *free* or *Skolem* constants, i.e., (first-order) constant symbols that do not occur in Σ. The set of models is extended correspondingly as follows. If k is one of the new constants and has type σ, for every model $\mathcal{I}$ of T and every $v \in \sigma^{\mathcal{I}}$, the extended theory has a model that interprets k as v and is otherwise identical to $\mathcal{I}$. It is not difficult to see that, for satisfiability purposes, free constants in a formula effectively behave like free variables.

2 Synthesis with SMT solvers

We are interested in synthesizing computable functions automatically from formal logical specifications stating properties of these functions. Under certain conditions described in the following, a version of the synthesis problem can be formulated 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 *first-order* Σ-formula[3] 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 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 $\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ and Σ-theories T where the T-validity of formulas of the form $\forall \boldsymbol{x}\, P[\lambda \boldsymbol{x}\, t, \boldsymbol{x}]$ is decidable.[4] In both approaches, we solve the synthesis conjecture by relying on quantifier-

[1] That is, every Σ-interpretation isomorphic to an interpretation in $\mathbf{I}$ is also in $\mathbf{I}$.

[2] For $T_1 \cup T_2$ to exist there must be models $\mathcal{I}_1$ of T_1 and $\mathcal{I}_2$ of T_2 that agree on the interpretation they give to the sort and function symbols shared by Σ_1 and Σ_2.

[3] In the sense that all occurrences of f in P are applied to some arguments.

[4] Note that since $P[f, \boldsymbol{x}]$ is first-order, all occurrences of $\lambda \boldsymbol{x}\, t$ in $P[\lambda \boldsymbol{x}\, t, \boldsymbol{x}]$ can be β-reduced.

Table 1 Scope of techniques presented

Conjectures\syntax	Unrestricted	Restricted
Single-invocation	Section 3	Section 5
Non-single invocation		Section 4

Each table entry shows the technique to use, depending on whether the conjecture is single-invocation or not, and depending on whether the solutions are required to conform to a syntax specifying a given subset of the space of solution functions

instantiation techniques to produce a first-order Σ-term $t[\boldsymbol{x}]$ of sort σ such that $\forall \boldsymbol{x}\ P[\lambda \boldsymbol{x}\ t, \boldsymbol{x}]$ is T-valid. When this t is found, the synthesized function is precisely $\lambda \boldsymbol{x}\ t$.

In principle, under the right assumptions on T, to determine the solvability of the conjecture $\exists f\ \forall \boldsymbol{x}\ P[f, \boldsymbol{x}]$ 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 since 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 $\boldsymbol{x}$ [17,40].

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 [13]. While these techniques are incomplete in general, they have been shown to be quite effective in practice [32,41]. For this reason, we advocate approaches to synthesis geared toward establishing the *unsatisfiability of the negation* of the synthesis conjecture:

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

We show in this paper how a syntactic solution $\lambda \boldsymbol{x}\ t$ 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 have 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. These methods are summarized in Table 1.

The first method (Sect. 3) applies to a restricted, but fairly common, case of synthesis problems $\exists f\ \forall \boldsymbol{x}\ P[f, \boldsymbol{x}]$ where the only applications of f in P are of the form $f(\boldsymbol{x})$. In this case, we can express the problem as a formula of the form $\forall \boldsymbol{x}\ \exists y\ Q[\boldsymbol{x}, y]$ with first-order quantifiers only, and then tackle its negation using appropriate quantifier instantiation techniques.

The second method (Sect. 4) applies to theories T with term-generated interpretations. This class of theories includes the majority of theories of interest in synthesis such as (versions of) the theories of integer, rational and floating point arithmetic, bit vectors, strings, constructible arrays, algebraic datatypes, finite sets, and their combinations.[5] This approach is well suited for the *syntax-guided synthesis* paradigm [2,3] where the synthesis conjecture is accompanied by an explicit syntactic restriction on the space of possible solutions. It is based on encoding

[5] These are versions that either do not have function symbols denoting partial functions or make those functions total in some way.

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.

Note that, in the case when no syntax restrictions are given and the problem has single-invocation form, the refutation-based approach avoids the need for encoding of syntax. This is the focus of Sect. 3; our system is particularly efficient in this case. For the case where syntax restrictions are required yet the problem is single-invocation we present two approaches in Sect. 5.

Assumption 1 For the rest of the paper, we fix a Σ-theory T and a class $\mathbf{P}$ of first-order quantifier-free Σ-formulas $P[f, \boldsymbol{x}]$ such that the T-satisfiability of formulas of the form $(\neg)P[\lambda\boldsymbol{x}\, t, \boldsymbol{x}]$ with t a Σ-term is decidable. We will consider synthesis conjectures for this theory from the set $\mathbf{L} := \{\exists f\, \forall \boldsymbol{x}\; P[f, \boldsymbol{x}] \mid P \in \mathbf{P}\}$.

Our canonical example of T throughout the paper will be the theory of integers. The signature of this theory contains only two sorts Int and Bool and the usual operators: the numerals, $+$, $*$, unary and binary $-$, $<$, and $\leq$ (with the last two of type $\mathsf{Int} \times \mathsf{Int} \rightarrow \mathsf{Bool}$). Let $\mathcal{I}$ be the Σ-interpretation that interprets the sort Int as the integers and interprets the various operators as expected. It is easy to see that this interpretation is term-generated. The models of T are all the Σ-interpretations that are isomorphic to $\mathcal{I}$. For this theory, $\mathbf{P}$ is effectively the class of all Boolean combinations of linear equations and inequations with one uninterpreted function f. Based on results originally by Presburger [36], one can easily argue that the T-satisfiability of formulas of the form $(\neg)P[\lambda\boldsymbol{x}\, t, \boldsymbol{x}]$, where $P \in \mathbf{P}$ and t is a linear term not containing f, is decidable. One can do that with current SMT solvers after eliminating from $P[\lambda\boldsymbol{x}\, t, \boldsymbol{x}]$ all occurrences of $\lambda\boldsymbol{x}\, t$ by β-reduction, i.e. $(\lambda\boldsymbol{x}\, t[\boldsymbol{x}])(\boldsymbol{s}) \rightsquigarrow t[\boldsymbol{s}]$.

3 Refutation-based synthesis

When axiomatizing properties of a desired function f of type $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$, a particularly well-behaved class are *single-invocation properties* (see, e.g., [19]). 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 [26–28]. 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.

Definition 1 A *single-invocation property* is any first-order formula of the form $Q[\boldsymbol{x}, f(\boldsymbol{x})]$ obtained by replacing y with $f(\boldsymbol{x})$ in a quantifier-free formula $Q[\boldsymbol{x}, y]$ not containing f.

Observe 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.[6] We will consider formulas like $Q[\boldsymbol{x}, f(\boldsymbol{x})]$ above that belong to the class $\mathbf{P}$ of formulas fixed in Assumption 1.[7] The conjecture $\exists f\, \forall \boldsymbol{x}\; Q[\boldsymbol{x}, f(\boldsymbol{x})]$ is logically equivalent to the *first-order* formula

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

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

[7] Note that $Q[\boldsymbol{x}, f(\boldsymbol{x})]$ is a formula over the variables f and $\boldsymbol{x}$ and so it has the form $P[f, \boldsymbol{x}]$ as in the definition of $\mathbf{P}$.

The equivalence is easy to see by observing that the conjecture is the Skolemized version of (3). By the semantics of $\forall$ and $\exists$, finding a model $\mathcal{I}$ for either formula 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 $\boldsymbol{e} \in \sigma_1^{\mathcal{I}} \times \cdots \times \sigma_n^{\mathcal{I}}$, the interpretation $\mathcal{I}[\boldsymbol{x} \mapsto \boldsymbol{e}, y \mapsto h(\boldsymbol{e})]$ satisfies $Q[\boldsymbol{x}, y]$. This section considers the case when **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 terms 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 a method for quantifier elimination [8,31]. Such cases have been examined in the context of software synthesis [27]. Here we propose instead an alternative, instantiation-based approach aimed at establishing the *un*satisfiability 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 free constants of the right sort. Finding a T-unsatisfiable finite set Γ of ground instances of $\neg Q[\mathbf{k}, y]$, 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 *Given $\boldsymbol{x} = (x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})$, let $Q[\boldsymbol{x}, y^{\sigma}]$ be a Σ-formula such that $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})] \in \mathbf{L}$ for some $f^{\sigma_1 \cdots \sigma_n \sigma}$ not occurring in $Q[\boldsymbol{x}, y]$. Suppose some set $\Gamma = \{\neg Q[\boldsymbol{x}, t_1], \ldots, \neg Q[\boldsymbol{x}, t_p]\}$, where $t_1[\boldsymbol{x}], \ldots, t_p[\boldsymbol{x}]$ are Σ-terms of sort σ, is T-unsatisfiable. Then, one solution for $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ (witness for f) is*

$$\lambda \boldsymbol{x}\, \mathsf{ite}(Q[\boldsymbol{x}, t_p], t_p, (\cdots \mathsf{ite}(Q[\boldsymbol{x}, t_2], t_2, t_1) \cdots)).$$

Proof Let ℓ be the solution specified above. Let $\mathcal{M}$ be any model of T and let $\boldsymbol{v} = (v_1, \ldots, v_n)$ be an arbitrary element of $\sigma_1^{\mathcal{M}} \times \cdots \times \sigma_n^{\mathcal{M}}$. It is enough to show that $\mathcal{I} \models Q[\boldsymbol{x}, \ell(\boldsymbol{x})]$ where $\mathcal{I} = \mathcal{M}[\boldsymbol{x} \mapsto \boldsymbol{v}]$. We have two cases.

(i) Suppose first that $\mathcal{I} \models Q[\boldsymbol{x}, t_i]$ for some $i \in \{2, \ldots, p\}$ and let m be the greatest such i. Then, by construction, $\ell(\boldsymbol{x})^{\mathcal{I}} = t_m^{\mathcal{I}}$, and thus $\mathcal{I} \models Q[\boldsymbol{x}, \ell(\boldsymbol{x})]$.

(ii) If, on the other hand, $\mathcal{I} \models \neg Q[\boldsymbol{x}, t_i]$ for all $i = 2, \ldots, p$, then $\ell(\boldsymbol{x})^{\mathcal{I}} = t_1^{\mathcal{I}}$. Since Γ is T-unsatisfiable by assumption, we have that $\neg Q[\boldsymbol{x}, t_2], \ldots, \neg Q[\boldsymbol{u}, t_p] \models_T Q[\boldsymbol{x}, t_1]$. It follows that $\mathcal{I} \models Q[\boldsymbol{x}, \ell(\boldsymbol{x})]$. □

Example 1 Let T be the theory of integer arithmetic as described in Sect. 2. Now consider the single-invocation 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 $\boldsymbol{x} = (x_1, x_2)$ where x_1 and x_2 are 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 single-invocation, 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

$\text{Synth}_{SI}(\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})])$:

1. Let $\mathsf{e}, \mathsf{k}_1, \ldots, \mathsf{k}_n$ be distinct fresh free constants where n is the size of $\boldsymbol{x}$
2. Let $\mathbf{k} = (\mathsf{k}_1, \ldots, \mathsf{k}_n)$ and $\Gamma := \emptyset$
3. While Γ is T-satisfiable
   If there is a T-model $\mathcal{I}$ of Γ satisfying $Q[\mathbf{k}, \mathsf{e}]$ then
      let $\Gamma := \Gamma \cup \{\neg Q[\mathbf{k}, t[\mathbf{k}]]\}$ for some Σ-term $t[\boldsymbol{x}]$ such that $t[\mathbf{k}]^{\mathcal{I}} = \mathsf{e}^{\mathcal{I}}$
   else
      return "no solution found"
4. Let $\{\neg Q[\mathbf{k}, t_1[\mathbf{k}]], \ldots, \neg Q[\mathbf{k}, t_p[\mathbf{k}]]\}$ be a T-unsatisfiable subset of Γ
5. 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 Synth_{SI} for single-invocation property $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$

instance, instantiate it with a_1 and then a_2 for y, producing the T-unsatisfiable set $\{\neg Q[\mathbf{a}, \mathsf{a}_1], \neg Q[\mathbf{a}, \mathsf{a}_2]\}$. Since a_1 and a_2 are fresh, it follows that $\{\neg Q[\boldsymbol{x}, x_1], \neg Q[\boldsymbol{x}, x_2]\}$ is T-unsatisfiable as well. 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. □

3.1 Synthesis by counterexample-guided quantifier instantiation

Given Proposition 1, the main question is how to get an 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 [32], 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., [22,46]), but with the major difference of being built directly into the SMT solver. The technique is illustrated by the procedure in Fig. 1, which grows a set Γ of ground instances of $\neg Q[\mathbf{k}, y]$. 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 satisfiable but all of its models falsify $Q[\mathbf{k}, \mathsf{e}]$. In the latter case, the search for a solution is 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.

3.2 Finding instantiations

The choice of the term $t[\boldsymbol{x}]$ 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, such as integer arithmetic, with a fixed interpretation for symbols in Σ and a distinguished set of

Table 2 A run of the procedure Synth_{SI} from Fig. 1 on input $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$, where $Q[\mathbf{a}, \mathsf{e}] := \mathsf{e} \geq \mathsf{a}_1 \wedge \mathsf{e} \geq \mathsf{a}_2 \wedge (\mathsf{e} \approx \mathsf{a}_1 \vee \mathsf{e} \approx \mathsf{a}_2)$

Step 3				
Iteration	Γ unsat?	$\Gamma \cup \mathsf{Q}[\mathbf{a}, \mathsf{e}]$ unsat?	$\mathcal{I} \models \Gamma \cup \mathsf{Q}[\mathbf{a}, \mathsf{e}]$	Add to Γ
1	No	No	$\{\mathsf{e} \mapsto 0, \mathsf{a}_1 \mapsto 0, \mathsf{a}_2 \mapsto 0\}$	$\neg Q[\mathbf{a}, \mathsf{a}_1]$
2	No	No	$\{\mathsf{e} \mapsto 1, \mathsf{a}_1 \mapsto 0, \mathsf{a}_2 \mapsto 1\}$	$\neg Q[\mathbf{a}, \mathsf{a}_2]$
3	Yes			

ground Σ-terms denoting the elements of a sort, a simple (if naive) choice for the term t in the procedure of Fig. 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 $t[\boldsymbol{x}]$, 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 implementation in the CVC4 solver [5] 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 a term $t[\boldsymbol{k}]$ among those in the equivalence class of e, other than e itself. This choice is correct because $t[\boldsymbol{k}]^{\mathcal{I}} = \mathsf{e}^{\mathcal{I}}$ by construction of $\mathcal{I}$.

Example 2 Consider the single invocation synthesis conjecture $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ where Q is defined in Eq. (6) from Example 1. An execution of the procedure from Fig. 1 on this input is illustrated in Table 2. In Step 1 and 2, we introduce the fresh free constants a_1, a_2, and e of sort Int and set Γ to the empty set. The columns of the table show details of the internal state of the procedure on iterations of Step 3. In the first iteration, we find a model $\mathcal{I}$ of $\Gamma \cup Q[\mathbf{a}, \mathsf{e}]$. Since $\mathcal{I}$ must satisfy the third conjunct of $Q[\mathbf{a}, \mathsf{e}]$, it must be the case that either $\mathsf{e}^{\mathcal{I}} = \mathsf{a}_1^{\mathcal{I}}$, or $\mathsf{e}^{\mathcal{I}} = \mathsf{a}_2^{\mathcal{I}}$, or both. Assume the model $\mathcal{I}$ is such that $\mathsf{e}^{\mathcal{I}} = \mathsf{a}_1^{\mathcal{I}} = \mathsf{a}_2^{\mathcal{I}} = 0$ on this iteration. Assuming our heuristic for instantiation selects a term whose interpretation in $\mathcal{I}$ is the same as e, we choose to add the formula $\neg Q[\mathbf{a}, \mathsf{a}_1]$ to Γ on this step. On the second iteration of Step 3 of the procedure, we determine that $\Gamma \cup Q[\mathbf{a}, \mathsf{e}]$ is still satisfiable. The model $\mathcal{I}$ on this iteration must satisfy $\neg Q[\mathbf{a}, \mathsf{a}_1]$, which is $\neg \mathsf{a}_1 \geq \mathsf{a}_1 \vee \neg \mathsf{a}_1 \geq \mathsf{a}_2 \vee (\mathsf{a}_1 \not\approx \mathsf{a}_1 \wedge \mathsf{a}_1 \not\approx \mathsf{a}_2)$ and simplifies to $\neg \mathsf{a}_1 \geq \mathsf{a}_2$. Notice that the solver can no longer choose to interpret $\mathsf{e}^{\mathcal{I}}$ as $\mathsf{a}_1^{\mathcal{I}}$ since it must satisfy $\mathsf{e} \geq \mathsf{a}_2$ and $\mathsf{a}_2 > \mathsf{a}_1$. Hence, the new $\mathcal{I}$ must be such that $\mathsf{e}^{\mathcal{I}} = \mathsf{a}_2^{\mathcal{I}}$ on this iteration. Subsequently, we add the formula $\neg Q[\mathbf{a}, \mathsf{a}_2]$ to Γ on this step, which can be simplified to $\neg \mathsf{a}_2 \geq \mathsf{a}_1$. Adding that constraint to Γ, which T-entails $\neg \mathsf{a}_1 \geq \mathsf{a}_2$, makes it T-unsatisfiable, causing the procedure to exit the while loop and terminate successfully. $\square$

The development of more sophisticated criteria for selecting instantiations that are both complete and efficient in practice is a subject of ongoing work [39]. For instance, a refinement of the above technique which considers the tightest lower (respectively, upper) bound for e in the current model $\mathcal{I}$ has been shown to be complete for linear arithmetic [39]. Quantifier elimination techniques [8,31] and approaches currently used to infer invariants from templates [11,29] can also be used for devising such criteria. 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 [2,3,37], 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 R directly into the SMT solver via a deep embedding [49,52] into the solver's logic of the terms meeting the restriction. 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.

Our approach is limited to theories T with term-generated interpretations and to restrictions R that can be expressed as datatypes, but is generic with respect to such theories and restrictions.

For simplicity, but without loss of generality, we assume that T has a set of function symbols $\approx^{\sigma\,\sigma\,\mathsf{Bool}}$ for all $\sigma \in \Sigma^{\mathrm{s}}$, $\neg^{\mathsf{Bool}\,\mathsf{Bool}}$, $\wedge^{\mathsf{Bool}\,\mathsf{Bool}\,\mathsf{Bool}}$, etc., corresponding to the various logical connectives and interpreted as expected in every model.[8] It is not difficult to prove that then every quantifier-free formula φ can be written equivalently as an equation $t_\varphi \approx \mathsf{tt}$ where t_φ is a Boolean term—that is, $\models_T \varphi \Leftrightarrow (t_\varphi \approx \mathsf{tt})$ for some term t_φ of type Bool. This is convenient as it allows us to treat function symbols and logical connectives uniformly. We will then sometimes abuse the notation and not distinguish between terms of type Bool and quantifier-free formulas.

Before defining T in its full generality, we introduce it with a concrete example.

4.1 An example

Consider again the synthesis conjecture (6) from Example 1, where T is the theory of linear integer arithmetic, but now with a syntactic restriction R for the solution space expressed by these mutually recursive datatypes:

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

The datatypes are meant to encode a term signature that includes nullary constructors for the integer variables x_1 and x_2 of (6), and constructors for the symbols of the arithmetic theory T. Terms of sort I (resp., B) refer to theory terms of sort Int (resp., Bool).

Instead of T, we now consider its combination T_{ev} with the theory of the datatypes above extended with two *evaluation operators*, that is, two function symbols $\mathsf{ev}^{\mathsf{I}\,\mathsf{Int}\,\mathsf{Int}\,\mathsf{Int}}$ and $\mathsf{ev}^{\mathsf{B}\,\mathsf{Int}\,\mathsf{Int}\,\mathsf{Bool}}$ respectively embedding I in Int and B in Bool. We define T_{ev} so that all of its models satisfy the formulas in Fig. 2. The evaluation operators effectively define an interpreter for programs (i.e., terms of sort I and B) with input parameters x_1 and x_2.

If the syntactic restriction R is expressed in the SyGuS language [37], it is possible to instrument an SMT solver that supports (user-defined) algebraic datatypes, quantifiers and linear arithmetic so that it constructs automatically from R both the datatypes I and B and the two evaluation operators. Reasoning about I and B can be done with the datatype subsolver as long as the solver is able to decide the satisfiability of ground formulas *as well as enumerate* their models.[9] Moreover, for each model $\mathcal{I}$ of a formula φ and each free variable x in φ, the

[8] That is, $\neg^{\mathsf{Bool}\,\mathsf{Bool}}$ interpreted as Boolean negation, $\wedge^{\mathsf{Bool}\,\mathsf{Bool}\,\mathsf{Bool}}$ as Boolean conjunction, $\approx^{\sigma\,\sigma\,\mathsf{Bool}}$ as the equality function, and so on.

[9] CVC4's solver for the theory of datatypes can do both things.

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

Fig. 2 Axiomatization of the evaluation operators in grammar R from Sect. 4.1

solver must be able to represent the value $x^{\mathcal{I}}$ as a *constructor term*, that is, a term containing only datatype constructor symbols. Reasoning about the evaluation operators is achieved by reducing ground terms of the form $\mathsf{ev}(d, t_1, t_2)$, where d is a constructor term, to smaller terms by using the axioms from Fig. 2 as rewrite rules, orienting each equation from left to right.

For instance, the formula $P[f, \boldsymbol{x}]$ in Eq. (5) from Example 1 can be restated in T_{ev} as the formula below where g is a variable of type I:

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

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

In the following we provide a general and formal definition of the theory T_{ev} mentioned in the example above, as well as a description of the synthesis procedure that we use with this theory.

4.2 From T to T_{ev}

Let T be the background Σ-theory fixed in Sect. 2 but with the additional assumptions that

1. T is the theory of some term-generated interpretation, and
2. its associated satisfiability procedure can also find a model for each formula φ it determines to be T-satisfiable, and output the values of φ's free variables in that model as ground Σ-terms.

The theory T_{ev} is an extension of T. Its construction depends on the syntax restriction R and on a tuple $\boldsymbol{x} = (x_1^{\sigma_1}, \ldots, x_k^{\sigma_k})$ of variables with $\sigma_1, \ldots, \sigma_k \in \Sigma^{\mathrm{s}}$. We fix such a tuple and consider conjectures of the form $\exists f\, \forall \boldsymbol{x}\, P[f, \boldsymbol{x}]$ from our conjecture language $\mathbf{L}$.

For generality, we do not discuss here any concrete language to specify the restriction R. Instead, we assume that it is possible to express R by a context-free grammar whose production rules can be faithfully encoded as a set of algebraic datatype definitions.[10] We use these datatypes to construct T_{ev}.

[10] We make this assumption to simplify the presentation. The SyGuS language [37] defines a more general class of restrictions R for which this is not necessarily possible.

The datatype restriction

Let D be a theory of algebraic datatypes [7] with signature Ω and set of constructor symbols Ω^{c} which shares no sort and no function symbols with T. We will denote D's constructors by c, possibly with subscripts. We call *constructor term* any term, possibly with variables, all of whose functions symbols are constructors.

We make the following assumptions on Ω:

1. There is a ground constructor term of type σ for each $\sigma \in \Omega^{\mathrm{s}}$.
2. The number of ground constructor terms of size n is finite for each $n > 0$.
3. There is a mapping $s : \Omega^{\mathrm{s}} \to \Sigma^{\mathrm{s}}$, an injective mapping $m : \Omega^{\mathrm{c}} \to \Sigma^{\mathrm{f}} \cup \boldsymbol{x}$, and an injective mapping $m : \mathbf{X} \to \mathbf{X}$ such that
   - for all $x \in \boldsymbol{x}$, there is a $c_x \in \Omega^{\mathrm{c}}$ with $m(c_x) = x$;
   - for all $c^{\delta_1 \cdots \delta_n \delta} \in \Omega^{\mathrm{c}}$, $m(c)$ has type $s(\delta_1) \times \cdots \times s(\delta_n) \to s(\delta)$;
   - for all $\delta \in \Omega^{\mathrm{s}}$ and $x \in \mathbf{X}_\delta$, $m(x) \in \mathbf{X}_{s(\delta)}$.

Intuitively, the datatype theory D captures the symbols and the syntax of some fragment of the language of quantifier-free Σ-formulas with variables in $\boldsymbol{x}$. Some sorts of T, possibly including Bool, are represented by one or more sorts of D; some function symbols f of T are represented in D by a constructor c_f whose type maps to the type of f; and every variable x in $\boldsymbol{x}$ is represented in D by a (constant) constructor c_x whose sort maps to the sort of x. Finally, every variable of datatype δ is mapped to a variable of the corresponding type $s(\delta)$ (other than those in $\boldsymbol{x}$). Note that the correspondence between sorts in D and sorts in T need not be one-to-one; this allows for greater flexibility in expressing language fragments. The fact that neither s nor m needs to be surjective corresponds to restricting the sorts and the function symbols in the fragment.

We will use m also to denote the homomorphic extension of the union of the two m's to constructor terms. Intuitively, this extension maps each constructor term (possibly with variables) to the term in T it represents.

Example 3 Looking back at the example in Sect. 4.1 we have that

$$
\begin{aligned}
\Omega^{\mathrm{s}} &= \{\mathsf{I},\ \mathsf{B}\}\\
\Omega^{\mathrm{c}} &= \{\mathsf{x}_1^{\mathsf{I}},\ \mathsf{x}_2^{\mathsf{I}},\ \mathsf{zero}^{\mathsf{I}},\ \mathsf{one}^{\mathsf{I}},\ \mathsf{plus}^{\mathsf{III}},\ \mathsf{minus}^{\mathsf{III}},\ \mathsf{if}^{\mathsf{BIII}},\ \mathsf{le}^{\mathsf{IIB}},\ \mathsf{eq}^{\mathsf{IIB}},\ \mathsf{and}^{\mathsf{BBB}},\ \mathsf{not}^{\mathsf{BB}}\}\\
s &= \{\mathsf{I} \mapsto \mathsf{Int},\ \mathsf{B} \mapsto \mathsf{Bool}\}\\
m &= \{\mathsf{x}_1 \mapsto x_1,\ \mathsf{x}_2 \mapsto x_2,\ \mathsf{zero} \mapsto 0,\ \ldots,\ \mathsf{if} \mapsto \mathsf{ite},\ \ldots,\ \mathsf{not} \mapsto \neg\}
\end{aligned}
$$

If d is the constructor term $\mathsf{le}(\mathsf{plus}(\mathsf{x}_1, z), \mathsf{plus}(\mathsf{x}_2, \mathsf{one}))$, say, where z is a variable of type I and $y = m(z)$, then $m(d) = x_1 + y \leq x_2 + 1$. □

The theory T_{ev}

The theory T_{ev} has a signature Σ_{ev} that extends the union of Σ and Ω with the following two families of new function symbols:

$$\{\mathsf{ev}^{\delta\,\sigma_1 \cdots \sigma_k\, s(\delta)} \mid \delta \in \Omega^{\mathrm{s}}\}$$
$$\{\mathsf{eq}^{\delta\,\delta\,\mathsf{Bool}} \mid \delta \in \Omega^{\mathrm{s}}\}$$

T_{ev} is the union of T, D and the theory consisting of all Σ_{ev}-interpretations that interpret the symbols $\mathsf{eq}^{\delta\delta\,\mathsf{Bool}}$ as the equality predicate and satisfy the following formulas for each $c^{\delta_1 \cdots \delta_n \delta} \in \Omega^{\mathrm{c}}$:

$$\forall z_1 \cdots z_k\, \mathsf{ev}(c, z_1, \cdots z_k) \approx z_i \text{ if } m(c) = x_i \text{ for some } i = 1, \ldots, k \tag{7}$$

$\mathrm{Synth}_{SG}(\exists g\, \forall \boldsymbol{x}\, P_{\mathsf{ev}}[g, \boldsymbol{x}])$:

1. Let $\Gamma = \emptyset$, let $n = 1$
2. Loop
   (a) While Γ is not n-satisfiable in T_{ev}
       If the n-unsatisfiability proof shows that Γ is actually T_{ev}-unsatisfiable then
         return "no solution"
       else
         increase n by some positive amount
   (b) Let $\mathcal{I}$ be a model of T_{ev} satisfying Γ with $size(g^{\mathcal{I}}) \le n$
   (c) If there is a model $\mathcal{J}$ of T_{ev} satisfying $\neg P_{\mathsf{ev}}[g^{\mathcal{I}}, \boldsymbol{x}]$ then
       let $\Gamma = \Gamma \cup \{P_{\mathsf{ev}}[g, \boldsymbol{t}]\}$ where $\boldsymbol{t}$ are ground Σ-terms such that $\boldsymbol{t}^{\mathcal{J}} = \boldsymbol{x}^{\mathcal{J}}$
     else
       return $g^{\mathcal{I}}$ as a solution

Fig. 3 A refutation-based syntax-guided synthesis procedure Synth_{SG} for $\exists g\, \forall \boldsymbol{x}\, P_{\mathsf{ev}}[g, \boldsymbol{x}]$. The expression $size(d)$ denotes the size of constructor term d

$$\forall z\, \mathsf{ev}(c(s_1, \ldots, s_n), z) \approx m(c)(\mathsf{ev}(s_1, z), \ldots, \mathsf{ev}(s_n, z)) \text{ if } m(c) \notin \boldsymbol{x} \tag{8}$$

One can show, although we will not do it here, that such union theory exists and is a *conservative extension* of its component theories, that is, for any Σ-formula φ and any Ω-formula ψ, (i) $\models_{T_{\mathsf{ev}}} \varphi$ iff $\models_T \varphi$ and (ii) $\models_{T_{\mathsf{ev}}} \psi$ iff $\models_D \varphi$.

Synthesis procedure

Our synthesis procedure applies to conjectures in the class

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

where instead of terms of the form $f(t_1, \ldots, t_k)$ in $\mathbf{P}$ we have, modulo β-reductions, terms of the form $\mathsf{ev}(g, t_1, , \ldots, t_k)$.

Let $\exists g\, \forall \boldsymbol{x}\, \varphi[g, \boldsymbol{x}] \in \mathbf{L}_2$ and observe that g is the only term of type $\delta \in \Omega^{\mathrm{s}}$ in φ and it occurs there only in terms of the form $\mathsf{ev}(d, \boldsymbol{t})$. We say that a set $\{\varphi[g, \boldsymbol{t}_1], \ldots, \varphi[g, \boldsymbol{t}_n]\}$ of instances of a formula φ like the above is *n-satisfiable in* T_{ev} if the set is satisfied by a model of T_{ev} that interprets g as a ground constructor term of size at most n. Our procedure relies on the following fact.

Proposition 2 *For any $n > 0$, n-satisfiability in T_{ev} is decidable.*

Proof Let $\{\varphi[g, \boldsymbol{t}_1], \ldots, \varphi[g, \boldsymbol{t}_n]\}$ be as in the definition of n-satisfiability. By our assumptions, there is a finite number of ground constructor terms of size at most n. For each such term d, each formula $\varphi[d, \boldsymbol{t}_i]$ can be effectively reduced to an T_{ev}-equivalent quantifier-free Σ-formula $\psi[\boldsymbol{t}_i]$ by using the axioms (7) and (8) as rewrite rules oriented from left to right. The satisfiability of $\{\psi[\boldsymbol{t}_1], \ldots, \psi[\boldsymbol{t}_n]\}$ can then be checked by the decision procedure for T. □

Given a synthesis conjecture $\exists g\, \forall \boldsymbol{x}\, P_{\mathsf{ev}}[g, \boldsymbol{x}] \in \mathbf{L}_2$ where g is the datatype variable standing for the program to be synthesized, we use a procedure analogous to that in Sect. 3 to extract a solution for g from a refutation of $\forall g\, \exists \boldsymbol{x}\, \neg P_{\mathsf{ev}}[g, \boldsymbol{x}]$. The main difference is that now g ranges over the datatype representing the restricted solution space.

The procedure is described in Fig. 3. It maintains a set Γ of ground instances of $P_{\mathsf{ev}}[g, \boldsymbol{x}]$ and an integer n representing an upper bound on the size of g. In each iteration of the main loop, it first determines if Γ is n-unsatisfiable. If it is, it might be because Γ itself is T_{ev}-unsatisfiable. In that case, the procedure has determined that the conjecture has no solution

Table 3 A run of the procedure Synth_{SG} from Fig. 3 on input $\exists g\,\forall \boldsymbol{x}\ P_{\mathsf{ev}}[g,\boldsymbol{x}]$, where $P_{\mathsf{ev}}[g,\boldsymbol{x}]$ is $\mathsf{ev}(g,\boldsymbol{x}) \geq x_1 \wedge \mathsf{ev}(g,\boldsymbol{x}) \geq x_2 \wedge (\mathsf{ev}(g,\boldsymbol{x}) \approx x_1 \vee \mathsf{ev}(g,\boldsymbol{x}) \approx x_2)$

Iteration	$\mathcal{I}$	$\mathcal{J}$	Added formula
1	$\{g \mapsto \mathsf{x}_1, \ldots\}$	$\{x_1 \mapsto 0, x_2 \mapsto 1, \ldots\}$	$P_{\mathsf{ev}}[g,0,1]$
2	$\{g \mapsto \mathsf{x}_2, \ldots\}$	$\{x_1 \mapsto 1, x_2 \mapsto 0, \ldots\}$	$P_{\mathsf{ev}}[g,1,0]$
3	$\{g \mapsto \mathsf{one}, \ldots\}$	$\{x_1 \mapsto 2, x_2 \mapsto 0, \ldots\}$	$P_{\mathsf{ev}}[g,2,0]$
4	$\{g \mapsto \mathsf{plus}(x_1, \mathsf{x}_2), \ldots\}$	$\{x_1 \mapsto 1, x_2 \mapsto 1, \ldots\}$	$P_{\mathsf{ev}}[g,1,1]$
5	$\{g \mapsto \mathsf{if}(\mathsf{le}(\mathsf{x}_1, \mathsf{one}), \mathsf{one}, \mathsf{x}_1), \ldots\}$	$\{x_1 \mapsto 1, x_2 \mapsto 2, \ldots\}$	$P_{\mathsf{ev}}[g,1,2]$
6	$\{g \mapsto \mathsf{if}(\mathsf{le}(\mathsf{x}_1, \mathsf{x}_2), \mathsf{x}_2, \mathsf{x}_1), \ldots\}$	none	

in the solution space defined by the restriction R and so it quits with failure. Otherwise, it has determined only that there are no solutions of size at most n and so it increments n. If the procedure finds that Γ is n-satisfiable in a model $\mathcal{I}$, it checks the T_{ev}-satisfiability of $\neg P_{\mathsf{ev}}[g^{\mathcal{I}}, \boldsymbol{x}]$. Note that since $g^{\mathcal{I}}$ is a ground constructor term, this check is effective, as explained in the proof of Proposition 2. If that formula has a model $\mathcal{J}$ this means that the candidate solution $g^{\mathcal{I}}$ has a *counterexample*, a tuple of input values, denoted by the terms $\boldsymbol{t}$, for which $g^{\mathcal{I}}$ fails to satisfy its specification P_{ev}. The procedure then adds the clause $P_{\mathsf{ev}}[g, \boldsymbol{t}^{\mathcal{J}}]$ to Γ, and repeats the main loop. That addition has the effect of requiring explicitly that all subsequent candidate solutions for g satisfy the specification P_{ev} for that counterexample for $g^{\mathcal{I}}$. Note that the terms $\boldsymbol{t}$ exist because T is the theory of a term-generated interpretation. If instead $\neg P_{\mathsf{ev}}[g^{\mathcal{I}}, \boldsymbol{x}]$ is T_{ev}-unsatisfiable, the procedure has determined that $g^{\mathcal{I}}$ satisfies $P_{\mathsf{ev}}[g^{\mathcal{I}}, \boldsymbol{x}]$ for all possible values of $\boldsymbol{x}$ and so it returns it as a solution.

The returned term $g^{\mathcal{I}}$ is a ground constructor term. A solution of the original conjecture $\exists f\,\forall \boldsymbol{x}\ P[f,\ \boldsymbol{x}]$ is then, by construction, the term $m(g^{\mathcal{I}})$.

We implemented the procedure Synth_{SG} in CVC4. Table 3 shows a run of the procedure for the conjecture from Sect. 4.1. We show the relevant values for g in the model $\mathcal{I}$ from Step (2a), and for x_1 and x_2 in Step (2c) for 6 iterations. Each successive model $\mathcal{I}$ interprets g as a term that meets the specification P_{ev} for all inputs $\boldsymbol{x}^{\mathcal{J}}$ found for previous candidates. After the sixth iteration, the procedure finds the candidate $\mathsf{if}(\mathsf{le}(\mathsf{x}_1, \mathsf{x}_2), \mathsf{x}_2, \mathsf{x}_1)$. For this term, the procedure cannot find an input value that falsifies P_{ev}, indicating that it is a solution for the synthesis conjecture.

Proposition 3 *The procedure* Synth_{SG} *has the following properties:*

1. *(Solution Soundness) If it returns a term d as a solution, then $\forall \boldsymbol{x}\ P_{\mathsf{ev}}[d, \boldsymbol{x}]$ is T_{ev}-satisfiable.*
2. *(Refutation Soundness) If it answers "no solution", then $\exists g\,\forall \boldsymbol{x}\ P_{\mathsf{ev}}[g, \boldsymbol{x}]$ is T_{ev}-unsatisfiable.*
3. *(Solution Completeness) it terminates (with a solution) whenever its input conjecture is satisfiable.*

Proof (1) Suppose the procedure returns some ground constructor term d as a solution. We have that $\neg P_{\mathsf{ev}}[d, \boldsymbol{x}]$ is T_{ev}-unsatisfiable. Thus, $\exists \boldsymbol{x}\ \neg P_{\mathsf{ev}}[d, \boldsymbol{x}]$ is T_{ev}-unsatisfiable and so $\forall \boldsymbol{x}\ P_{\mathsf{ev}}[d, \boldsymbol{x}]$ is T_{ev}-satisfiable.

(2) Suppose the procedure returns "no solution". Then, there is a set $\Gamma = \{P_{\mathsf{ev}}[g, \boldsymbol{t}_1], \ldots, P_{\mathsf{ev}}[g, \boldsymbol{t}_p]\}$ that is T_{ev}-unsatisfiable, where $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_p$ are tuples of ground terms. This implies that $\forall \boldsymbol{x}\ P_{\mathsf{ev}}[g, \boldsymbol{x}]$ is T_{ev}-unsatisfiable, making $\exists g\,\forall \boldsymbol{x}\ P_{\mathsf{ev}}[g, \boldsymbol{x}]$ T_{ev}-unsatisfiable as well.

(3) Suppose $\exists g\, \forall \boldsymbol{x}\; P_{\mathsf{ev}}[g, \boldsymbol{x}]$ is T_{ev}-satisfiable. Then there is a ground constructor term d_0 of minimal size n_0 such that $\forall \boldsymbol{x}\; P_{\mathsf{ev}}[d_0, \boldsymbol{x}]$ is T_{ev}-satisfiable. Observe first that all the satisfiability tests in the procedure are effective. In particular, the one in Step (2a) is so by Proposition 2; the one in Step (2c) because $g^{\mathcal{I}}$ contains no variables. Hence it is enough to show that both loops in the procedure are terminating.

The while loop in Step (2a) terminates because the n counter is incremented at every iteration of the while loop. If the loop does not terminate for another reason, the value of n will eventually reach some value $k \geq n_0$, after which the loop condition will fail. This is because Γ is a consequence of $\exists g\, \forall \boldsymbol{x}\; P_{\mathsf{ev}}[g, \boldsymbol{x}]$, and hence by assumption is satisfied by a model $\mathcal{I}$ where $g^{\mathcal{I}}$ is d_0, whose size is at most n_0 and hence, by definition of n-satisfiability, Γ is k-satisfiable in T_{ev}.

The main loop terminates because, once $n = k$, by our assumptions on the datatype signature Ω, the set S of ground constructor terms with size $\leq k$ is finite. The subset S_0 of S of terms such that $\forall \boldsymbol{x}\; P_{\mathsf{ev}}[d, \boldsymbol{x}]$ is T_{ev}-satisfiable is non-empty. The value of $g^{\mathcal{I}}$ on each iteration in Step (2b) is unique. This is because each d chosen on prior iterations is such that Γ contains $P_{\mathsf{ev}}[g, \boldsymbol{t}]$ for some $\boldsymbol{t}$ where $\neg P_{\mathsf{ev}}[d, \boldsymbol{t}]$ is T_{ev}-satisfiable. By our assumptions on T, this implies all models $\mathcal{I}$ of Γ are such that $g^{\mathcal{I}} \neq d$. Hence, a model $\mathcal{I}$ where $g^{\mathcal{I}}$ with $g^{\mathcal{I}} \in S_0$ will be eventually chosen in Step (2b). At that point, the test in Step (2c) will fail and $g^{\mathcal{I}}$ will be returned. □

Note the procedure is not refutation complete: it can diverge if, but only if, the input synthesis conjecture has no solution.

4.3 Early pruning by symmetry breaking

The procedure Synth_{SG} in Fig. 3 effectively considers multiple candidate solutions for the negated synthesis conjecture $\forall \boldsymbol{x}\; \neg P_{\mathsf{ev}}[g, \boldsymbol{x}]$. A concrete implementation of the procedure can be obtained through a form of branch and bound search, where the bound is on the size of the ground constructor terms representing candidate solutions. For efficiency, it is important for such an implementation to avoid spending time considering candidate solutions that are equivalent to one another. Our implementation of the procedure within CVC4 uses *blocking clauses* to express at some point that all remaining candidate solutions for g in the current branch of the search are equivalent to previously considered solutions, forcing the search of that branch to be abandoned. Adding these clauses to the problem can significantly increase performance.

This is typical in SMT solvers based, like CVC4, on the DPLL(T) architecture [33]. Such solvers check the satisfiability of a quantifier-free formula φ in a background Σ-theory T by combining a SAT solver, used to check the propositional satisfiability φ, with a (set of) *theory solvers* for checking the T-satisfiability of an evolving set M of Σ-literals representing a (partial) truth value assignment for the atoms of φ. If at any time a theory solver determines a subset C of M to be T-unsatisfiable, it adds the *blocking clause* $\bigvee_{l \in C} \neg l$ to the SAT solver, indicating that at least one literal in M must be retracted. Blocking clauses, in addition to expressing that the current set M is T-unsatisfiable, can also be used to prune future search branches by prohibiting a certain combination of literals to be reasserted again in the set M. The same mechanism is used in some advanced approaches to SAT and SMT also to break *symmetries* [1,12]. We can understand symmetries informally here as the existence of several solutions for a subproblem which are, however, equivalent for all interesting purposes.

Symmetries arise in our procedure Synth_{SG} because distinct constructor terms in T_{ev} often represent terms in T that are T-equivalent. For instance, in the example of Sect. 4.1

the arithmetic terms $x_1 + x_2$, $x_2 + x_1$, and $x_1 + (0 + x_2)$, corresponding respectively to the constructor terms $\mathsf{plus}(\mathsf{x}_1, \mathsf{x}_2)$, $\mathsf{plus}(\mathsf{x}_2, \mathsf{x}_1)$, and $\mathsf{plus}(\mathsf{x}_1, \mathsf{plus}(\mathsf{zero}, \mathsf{x}_2))$ are all T-equivalent. This means that once, $\mathsf{plus}(\mathsf{x}_1, \mathsf{x}_2)$, say, is considered as a candidate solution the others should not because they fundamentally represent the same solution.

Breaking symmetries To discuss how we break symmetries in our implementation of Synth_{SG} let us start with the following observation. First, for each constructor $c^{\delta_1 \cdots \delta_n \delta} \in \Sigma^{\mathsf{c}}_{\mathsf{ev}}$ and $i = 1, \ldots, n$ let $\mathsf{s}^{\delta\delta_i}_{c,i}$ be the *selector* for the i^{th} argument of c. Also, let $\mathsf{is}^{\delta\,\mathsf{Bool}}_c$ denote the corresponding *tester*.[11] Then, a set of Σ_{ev}-literals M entails part of the shape that the values of a datatype variable g must have in all models of M. For instance, if $M = \{\mathsf{is}_{\mathsf{plus}}(g), \mathsf{is}_{\mathsf{zero}}(\mathsf{s}_{\mathsf{plus},1}(g))\}$, then all the T_{ev}-models of M must interpret g as a constructor term of the form $\mathsf{plus}(\mathsf{zero}, d)$ for some d. We will write $\mathsf{d}_{g,M}[z]$ to denote the smallest constructor term such that each variable of z occurs in the term at most once and $M \models_{T_{\mathsf{ev}}} \exists z\, \mathsf{d}_{g,M} \approx g$. Intuitively, $\mathsf{d}_{g,M}$ is a template that expresses everything that M entails about the shape of g, and nothing more. The idea is to consider only terms $\mathsf{d}_{g,M}$ whose analogues $m(\mathsf{d}_{M,g})$ in T are unique up to *equivalence*, for some suitable equivalence relation $\equiv$.

To define $\equiv$ we assume that every Σ-term t can be effectively reduced to some T-equivalent, unique and irreducible term, which we denote by $t{\downarrow}$ and call a *normal form* of t.[12] Then two Σ-terms t and u are $\equiv$-equivalent if there is a bijective renaming ρ of the variables of $u{\downarrow}$ such that $t{\downarrow}$ and $\rho(u{\downarrow})$ are T-equivalent.

When checking T_{ev}-satisfiability, we maintain a set of terms $\mathcal{S}$ indicating the shapes of terms we have considered (or are currently considering) for a datatype variable g so far, and a set $\mathcal{N}$ containing the normal forms for all terms in $\mathcal{S}$. At any time we are asked to check the T_{ev}-satisfiability of a set M with a free datatype variable g and $m(\mathsf{d}_{g,M})$ is not in $\mathcal{S}$, we add that term to $\mathcal{S}$ and construct the term $u = m(\mathsf{d}_{g,M}){\downarrow}$. If u is $\equiv$-equivalent to a term $m(\mathsf{d}_{g,M_0}){\downarrow}$ in $\mathcal{N}$ then we produce the blocking clause $\bigvee_{l \in C} \neg l$, for some minimal $C \subseteq M$ such that $\mathsf{d}_{g,C} = \mathsf{d}_{g,M}$. Otherwise, we add u to $\mathcal{N}$.

In this method, we use the same normal form $t{\downarrow}$ for terms t that is used by theory solver for T in CVC4, making this method parametric in the background theory considered.

Examples of blocking clauses are shown in Table 4. The first clause assumes that we have already determined that there are no solutions where g is interpreted as a term of the form $(z_1' + z_2') + z_3'$ for any z_1', z_2', z_3'. It is not useful to consider the case where g is a term of the form $z_1 + (z_2 + z_3)$, by noting their normal forms are equivalent up to renaming of variables. More generally, this scheme restricts our search to consider only left-associated chains of applications of associative operators. The other cases are similar, ensuring that the solver avoids solutions involving addition with 0, sums of monomials with identical normal forms, and terms where Boolean simplification results in a previously considered term.

5 Single invocation techniques for syntax-guided problems

In this section, we consider the combined case of *single-invocation synthesis conjectures with syntactic restrictions* R, giving two alternative solving methods.

[11] The selector $\mathsf{s}_{c,i}$ is such that $\models_D \mathsf{s}_{c,i}(c(d_1, \ldots, d_n)) \approx d_i$ for all constructor terms $d_1, \ldots, d_n$ of respective type $\delta_1, \ldots, \delta_n$. The tester is_c is such that, for all terms d of type δ, $\models_D \mathsf{is}_c(d) \approx \mathsf{tt}$ iff the top symbol of d is c.

[12] Note that this assumption is pragmatic and can be made arbitrarily mild. For instance, depending on the theory, in the worst case one can always consider every term to be irreducible.

Table 4 Examples of symmetry breaking clauses for the grammar from the example in Sect. 4.1

$m(\mathsf{d}_{g,M_0}) \in \mathcal{S}$	$m(\mathsf{d}_{g,M_0})\downarrow \in \mathcal{N}$	$m(\mathsf{d}_{g,M})$	$m(\mathsf{d}_{g,M})\downarrow$	Blocking clause
$(z_1' + z_2') + z_3'$	$z_1' + z_2' + z_3'$	$z_1 + (z_2 + z_3)$	$z_1 + z_2 + z_3$	$\neg(\mathsf{is}_{\mathsf{plus}}(g) \wedge \mathsf{is}_{\mathsf{plus}}(\mathsf{s}_{\mathsf{plus},2}(g)))$
z_1'	z_1'	$0 + z_1$	z_1	$\neg(\mathsf{is}_{\mathsf{plus}}(g) \wedge \mathsf{is}_{\mathsf{zero}}(\mathsf{s}_{\mathsf{plus},1}(g)))$
$x_1 + x_2$	$x_1 + x_2$	$x_2 + x_1$	$x_1 + x_2$	$\neg(\mathsf{is}_{\mathsf{plus}}(g) \wedge \mathsf{is}_{\mathsf{x_2}}(\mathsf{s}_{\mathsf{plus},1}(g)) \wedge \mathsf{is}_{\mathsf{x_1}}(\mathsf{s}_{\mathsf{plus},2}(g)))$
$\mathsf{ite}(z_1', z_2', z_3')$	$\mathsf{ite}(z_1', z_2', z_3')$	$\mathsf{ite}(\neg z_1, z_2, z_3)$	$\mathsf{ite}(z_1, z_3, z_2)$	$\neg(\mathsf{is}_{\mathsf{ite}}(g) \wedge \mathsf{is}_{\mathsf{not}}(\mathsf{s}_{\mathsf{ite},1}(g)))$

Assuming there are no models for M_0, we conclude there are no models for M

Table 5 A run of the procedure Synth_{SI} from Fig. 1 on input $\exists g\,\forall \boldsymbol{x}\; Q_{\mathsf{B}}[\boldsymbol{x}, g(\boldsymbol{x})]$, where $Q_{\mathsf{B}}[\boldsymbol{a}, \mathsf{e}]$ is $\mathsf{ev}(\mathsf{and}(\mathsf{le}(\mathsf{x}_1, \mathsf{e}), \mathsf{and}(\mathsf{le}(\mathsf{x}_2, \mathsf{e}), \mathsf{or}(\mathsf{eq}(\mathsf{e}, \mathsf{x}_1), \mathsf{eq}(\mathsf{e}, \mathsf{x}_2)))), \boldsymbol{a})$

Iteration	Γ unsat?	$\Gamma \cup Q_{\mathsf{B}}[\mathbf{a}, \mathsf{e}]$ unsat?	$\mathcal{I} \models \Gamma \cup Q_{\mathsf{B}}[\mathbf{a}, \mathsf{e}]$	Add to Γ
1	No	No	$\{\mathsf{e} \mapsto \mathsf{x}_1, \mathsf{a}_1 \mapsto 0, \mathsf{a}_2 \mapsto 0\}$	$\neg Q_{\mathsf{B}}[\mathbf{a}, \mathsf{x}_1]$
2	No	No	$\{\mathsf{e} \mapsto \mathsf{x}_2, \mathsf{a}_1 \mapsto 0, \mathsf{a}_2 \mapsto 1\}$	$\neg Q_{\mathsf{B}}[\mathbf{a}, \mathsf{x}_2]$
3	Yes			

The first method uses the deep embedding introduced in Sect. 4 to encode the first-order variant of the synthesis conjecture. This approach handles the exploration of different cases *natively* within the SMT solver while explicitly modeling the syntax of terms returned in each case using the embedding.

The second method splits the synthesis process into two steps, where we first solve the synthesis conjecture while ignoring its associated syntactic restrictions, and afterwards reconstruct an equivalent solution meeting them.

To simplify the exposition we restrict ourselves to sets R of syntactic restrictions expressed by a datatype I for programs and a datatype B for Boolean expressions. The general case of syntactic restrictions with more datatypes is similar.

5.1 Method 1: Encode single invocation property in T_{ev}

We consider first the case in which both of the following two conditions hold:

1. S contains the constructor $\mathsf{if}^{\mathsf{BIII}}$, mapped to the if-then-else logical operator ite, and
2. the function to be synthesized is specified by a single-invocation property that can be expressed as a term of sort B.

This is the case for the syntactic restrictions R from the example in Sect. 4.1. To solve this conjecture, we may encode the synthesis conjecture into the language of R, and use the procedure Synth_{SI} from Fig. 1, since by the above properties of R it is guaranteed to find solutions meeting our syntactic requirements.

Example 4 Consider the single invocation property $\exists f\,\forall \boldsymbol{x}\; Q[\boldsymbol{x}, f(\boldsymbol{x})]$ where Q is defined in Eq. (6) from Example 1, where f has type $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$. Assume the syntactic restrictions R from Sect. 4.1 are given for solutions of this conjecture. We may rephrase the conjecture as $\exists g\,\forall \boldsymbol{x}\; Q_{\mathsf{B}}[\boldsymbol{x}, g(\boldsymbol{x})]$, where

$$Q_{\mathsf{B}}[\boldsymbol{x}, y] := \mathsf{ev}(\mathsf{and}(\mathsf{le}(\mathsf{x}_1, y), \mathsf{and}(\mathsf{le}(\mathsf{x}_2, y), \mathsf{or}(\mathsf{eq}(y, \mathsf{x}_1), \mathsf{eq}(y, \mathsf{x}_2)))), \boldsymbol{x}) \tag{9}$$

g is a function of type $\mathsf{Int} \times \mathsf{Int} \to \mathsf{I}$, and y is of type I where I is a datatype that encodes integer terms that meet the syntactic restrictions R. A run of the procedure Synth_{SI} from Fig. 1 on input $\exists g\,\forall \boldsymbol{x}\; Q_{\mathsf{B}}[\boldsymbol{x}, g(\boldsymbol{x})]$ is shown in Table 5.

In Step 1, we initialize Γ to $\emptyset$, and introduce the fresh free constants $\mathsf{a}_1, \mathsf{a}_2$ of sort Int, with $\mathbf{a} = (\mathsf{a}_1, \mathsf{a}_2)$, and a fresh constant e of sort I. On the first iteration of Step 2 of the procedure, we find that Γ is satisfiable, and that $\Gamma \cup Q_{\mathsf{B}}[\mathbf{a}, \mathsf{e}]$ has a model $\mathcal{I}$. In contrast to the run described in Table 2 where e was of sort Int, the e in Table 5 is of sort I. Because of this, the heuristic used in Table 2, which considered the relation between terms of sort Int, is no longer applicable for choosing an instance to add to Γ. Instead, we use a heuristic that adds instances to Γ based on the *value* of e in model $\mathcal{I}$, which interprets the datatype I as the set of terms built from its constructors. One such model interprets e as the term x_1. Assuming this model, we add the instance $\neg Q_{\mathsf{B}}[\mathsf{a}, \mathsf{x}_1]$ to Γ, after which we discover that $\Gamma \cup Q_{\mathsf{B}}[\mathsf{a}, \mathsf{e}]$ is satisfiable. Assuming the model on the next iteration interprets e as x_2, we add the instance $\neg Q_{\mathsf{B}}[\mathsf{a}, \mathsf{x}_2]$ to Γ, which, together with the previous instance, makes Γ unsatisfiable. This tells us that the solution

$$\lambda x_1\, \lambda x_2\, \mathsf{ite}(\varphi[x_1, x_2], \mathsf{x}_1, \mathsf{x}_2) \tag{10}$$

where

$$\varphi[x_1, x_2] = \mathsf{ev}(\mathsf{and}(\mathsf{le}(\mathsf{x}_1, \mathsf{x}_1), \mathsf{and}(\mathsf{le}(\mathsf{x}_2, \mathsf{x}_1), \mathsf{or}(\mathsf{eq}(\mathsf{x}_1, \mathsf{x}_1), \mathsf{eq}(\mathsf{x}_1, \mathsf{x}_2)))), x_1, x_2)$$

is a solution for g in $\exists g\, \forall \boldsymbol{x}\, Q_{\mathsf{B}}[\boldsymbol{x}, g(\boldsymbol{x})]$. The analogue of this solution in the language of T, namely

$$\lambda x_1\, \lambda x_2\, \mathsf{ite}(x_1 \leq x_1 \wedge x_2 \leq x_1 \wedge (x_1 \approx x_1 \vee x_1 \approx x_2), x_1, x_2) \tag{11}$$

is a solution for f in $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ that meets the syntactic restrictions R. $\square$

In contrast to the solutions returned by procedure Synth_{SG}, all solutions returned by the procedure Synth_{SI} have the same high-level structure: they are case-based functions with the various cases separated by (possibly nested) ite's. One advantage of running Synth_{SI} in the way described in the example above then is that only the individual cases need to be synthesized since the conditions in the ite's come directly from the input conjecture. Moreover, Synth_{SI} maintains the benefits of the theoretical properties stated in Proposition 3, given a fair strategy for selecting candidate terms for instantiation. However, the selection criterion for instantiation in Table 5 is now much weaker than the one used in Table 2. In Table 5, we assumed the procedure interpreted e as x_1 and x_2 on the two iterations of the run. But it may have interpreted, say, e as zero on the first iteration, or one on the second iteration. Assuming a fair strategy for enumerating solutions for e_1, the number of iterations of Step 2 of the procedure could have been up to four instead.

Since the efficiency of our approach for synthesis is highly dependent upon having a good heuristics for selecting instances to add to Γ, we describe an alternative method in the following.

5.2 Method 2: Solve single invocation property, reconstruct solution in *R*

An alternative approach to solve single-invocation synthesis conjectures with syntactic restrictions R is to run the procedure Synth_{SI} from Fig. 1 as is, ignoring the restrictions, and subsequently reconstruct from its returned solution one that satisfies them. This has two significant advantages over the method described in Sect. 5.1. First, reasoning about conjectures directly allows us more powerful criteria for selecting instantiations, as seen in the differences between the runs in Tables 2 and 5. Second, our experimental evaluation found that the overhead of solving ground satisfiability problems that involve an embedding into datatypes for syntax-guided problems is significant with respect to the performance of the solver on problems with no syntactic restrictions.

Figure 4 presents a procedure, called reconstruct, that takes as input a term t and a datatype I, and attempts to construct a term that is equivalent to t and meets the syntactic restrictions specified by I. This procedure maintains an evolving set A of triples of the form $(u{\downarrow}, u, \delta)$, where δ is the datatype I or B, u is a term satisfying the restrictions specified by δ, and $\downarrow$ is the normalization operator as described in Sect. 4.3. The procedure incrementally calls the (non-deterministic) subprocedure rcon, which takes a normal form term t, a datatype δ and the set A above, and returns a pair (u, U) where u is a term equivalent to the input t to reconstruct(t, δ):

$A := \emptyset$
$t' := t\!\downarrow$
for $i = 1, 2, \ldots$
  $(u, U) := \text{rcon}(t', \delta, A)$;
  if U is empty, return u;
  otherwise, for each datatype δ_j occurring in U:
    $A := A \cup \{\, (m(d_i)\!\downarrow,\ m(d_i),\ \delta_j)\,\}$,
    where d_i is the i^{th} term in a fair enumeration of the elements of δ_j

rcon(t, δ, A):

if $(t, u, \delta) \in A$, return $(u, \emptyset)$; otherwise, do one of the following:
 (a) choose a $f(t_1, \ldots, t_n)$ s.t. $f(t_1, \ldots, t_n)\!\downarrow\ = t$ and $m(c) = f$ for some $c^{\delta_1 \cdots \delta_n \delta}$ in δ
  let $(u_i, U_i) = \text{rcon}(t_i\!\downarrow, \delta_i, A)$ for $i = 1, \ldots, n$
  return $(f(u_1, \ldots, u_n), U_1 \cup \ldots \cup U_n)$
 (b) return $(t, \{(t, \delta)\})$

Fig. 4 A procedure reconstruct for finding a term equivalent to t that meets the syntactic restrictions specified by datatype δ

reconstruct, and U is a set of pairs of the form (u', δ') where u' is a subterm of u that fails to satisfy the syntactic restriction expressed by datatype δ'. The procedure rcon may either try to match t to a term whose top symbol f has an analogue c in δ, or simply return the set $\{(t, \delta)\}$, indicating that it failed to match t to the syntactic restriction given by δ. Overall, the procedure alternates between calling rcon and adding triples to A until rcon(t, δ, A) returns a pair of the form $(s, \emptyset)$, indicating that u is a term T-equivalent to t that satisfies the syntactic restrictions embodied by I and B.

Example 5 Consider the single invocation property $\exists f\, \forall \boldsymbol{x}\, Q[\boldsymbol{x}, f(\boldsymbol{x})]$ where Q is defined in Eq. (6) from Example 1, assume the syntactic restrictions R from Sect. 4.1 are given for solutions of this conjecture. Say we use the procedure Synth$_{SI}$ from Fig. 1 for finding a solution to this conjecture, and that Synth$_{SI}$ returns the solution $\lambda \boldsymbol{x}\, u$ for f, where $u = \mathsf{ite}((-1 * x_1) + x_2 \leq 0, x_1, x_2)$. Note that this is indeed a solution for our conjecture, but it does not meet the syntactic restrictions given by the datatype I in R, since it contains the multiplication operator $*$ and unary minus $-$. To construct from that a solution that meets the syntactic restrictions represented by I, we run the procedure reconstruct from Fig. 4 on u and I. We let $A = \emptyset$, and call rcon$(u\!\downarrow, \mathsf{I}, A)$. The intermediate calls of a run of rcon$(u, \mathsf{I}, \emptyset)$ are shown below, where we assume that $u'\!\downarrow = u'$ for all subterms u' of u.

t	δ	return
$\mathsf{ite}((-1 * x_1) + x_2 \leq 0, x_1, x_2)$	I	$(\mathsf{ite}((-1 * x_1) + x_2 \leq 0, x_1, x_2), \{(-1 * x_1, \mathsf{I})\})$
$(-1 * x_1) + x_2 \leq 0$	B	$((-1 * x_1) + x_2 \leq 0, \{(-1 * x_1, \mathsf{I})\})$
x_1	I	$(x_1, \emptyset)$
x_2	I	$(x_2, \emptyset)$
0	I	$(0, \emptyset)$
$(-1 * x_1) + x_2$	I	$((-1 * x_1) + x_2, \{(-1 * x_1, \mathsf{I})\})$
$(-1 * x_1)$	I	$(-1 * x_1, \{(-1 * x_1, \mathsf{I})\})$

In more detail, on the initial call to rcon, since A is empty and ite is the analogue of constructor $\mathsf{if}^{\mathsf{BII}}$ in I, the procedure rcon may choose to return a pair based on the result of calling rcon$((-1 * x_1) + x_2 \leq 0, \mathsf{B}, A)$, rcon$(x_1, \mathsf{I}, A)$, and rcon$(x_2, \mathsf{I}, A)$. We may similarly traverse the function symbols of all subterms of u, with the exception of $(-1 * x_1)$. On the recursive call to rcon$(-1 * x_1, \mathsf{I}, \emptyset)$, the procedure chooses to returns a pair whose second component contains $(-1 * x_1, \mathsf{I})$, indicating it failed to produce a solution that met our syntactic restrictions. Overall, $\text{rcon}(u, \mathsf{I}, \emptyset)$ returns the pair $(u, \{(-1 * x_1, \mathsf{I})\})$. Since the second component is non-empty, in the procedure reconstruct, we enumerate the first element of I, x_1 say, whose analogue $m(\mathsf{x}_1)$ in T is x_1, and add the triple (x_1, x_1, I) to A. This indicates that there is a term in I (as witnessed by x_1) that is equivalent to x_1. The call to $\text{rcon}(u, \mathsf{I}, A)$ will again return the same value, after which we pick another triple to add to A and repeat. This process continues until we enumerate the term $\mathsf{minus}(\mathsf{x}_2, \mathsf{x}_1)$ say, whose analogue $m(\mathsf{minus}(\mathsf{x}_2, \mathsf{x}_1))$ in T is $x_2 - x_1$. Assume $(x_2 - x_1){\downarrow} = (-1 * x_1) + x_2$. We add the pair $((-1{*}x_1){+}x_2, x_2{-}x_1, \mathsf{I})$ to A. After doing so, the subcall to $\text{rcon}((-1{*}x_1){+}x_2, \mathsf{I}, A)$ returns $(x_2 - x_1, \emptyset)$, and hence $\text{rcon}(s, \mathsf{I}, A)$ may return $(\mathsf{ite}(x_2 - x_1 \leq 0, x_1, x_2), \emptyset)$. This indicates that $\lambda x_1 x_2\, \mathsf{ite}(x_2 - x_1 \leq 0, x_1, x_2)$ is equivalent to $\lambda x_1 x_2\, \mathsf{ite}((-1 * x_1) + x_2 \leq 0, x_1, x_2)$ and thus is a solution for our conjecture, and moreover meets the restrictions specified by I. □

The procedure reconstruct depends upon a normal form for terms. Since the top symbol of t is generally ite, this normalization includes low-level rewriting of literals within t as well as high-level rewriting steps such as ite simplification, redundant subterm elimination and destructive equality resolution. Also, notice that we are not insisting that every two T-equivalent terms have the same normal form, and thus normal forms only under-approximate T-equivalence between terms. Stronger term reduction mechanisms that reduce larger sets of terms to the same normal form yield tighter under-approximations of T-equivalence, thus improving the performance of the reconstruction. This is the case for theories such as linear arithmetic whose normal form for terms is a sorted list of monomials, as opposed for instance to theories such as bitvectors, where distinct terms in normal form may be still equivalent to one another.

We use several optimizations, omitted in the description of the procedure in Fig. 4, to increase the likelihood that the procedure terminates quickly. 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 whose nodes are triples of the form (t, δ, f), where t is a term, δ is a datatype, and f is either a function (in which case we call it a *complete* node) or a distinguished symbol $\oslash$ (in which case we call it an *incomplete* node). In this graph, complete nodes have children of the form $(t_1, \delta_1, f_1), \dots, (t_n, \delta_n, f_n)$, where $f(t_1, \dots, t_n){\downarrow} = t$ and there is a $c^{\delta_1 \dots \delta_n \delta}$ in δ such that $m(c) = f$, and incomplete nodes have no children. We then enumerate datatype terms d_i for all datatypes δ that occur in incomplete nodes $(t, \delta, \oslash)$, replacing such a node with a corresponding graph containing only complete nodes if we find a d_i such that $m(d_i){\downarrow} = t$. We succeed in finding a term t that meets the syntactic restrictions specified by datatype δ if we construct a graph of this form with root node (t, δ, f) that contains no incomplete nodes, where the solution can be extracted by traversing this graph.

Another important optimization is that our implementation simultaneously tries multiple alternatives considering a term t with restrictions δ. For instance, in Example 5, when calling rcon on $(-1{*}x_1){+}x_2 \leq 0$ and B, we chose to match it to $(-1{*}x_1){+}x_2 \leq 0$ whose top symbol $\leq$ has an analogue le in B. However, we may have chosen to match it to $\neg((-1{*}x_1){+}x_2 > 0)$ instead, noting not is also in B. In terms of the directed acyclic graph described above, we introduce multiple nodes of the form $(t, \delta, f_1), \dots, (t, \delta, f_n)$ when considering a term t with restrictions δ. These nodes are chosen in a way such that the size of the graph does not diverge. Whenever the graph whose root node is (t, δ, f_i) contains only complete nodes for *any* i, we remove the nodes (t, δ, f_j) for $j \neq i$ from the graph, direct their inward edges to (t, δ, f_i), and prune their children accordingly. Again, we succeed for t and δ when we construct a graph with root node (t, δ, f) containing no incomplete nodes.

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

As mentioned, we implemented the techniques from the previous sections in CVC4 [5], an open-source SMT solver with support for quantified formulas and a wide range of theories including arithmetic, bitvectors, and algebraic datatypes.[13] We considered multiple configurations of CVC4 corresponding to the techniques mentioned in this paper. Configuration **cvc4+sg** executes the syntax-guided procedure from Sect. 4, even in cases where the synthesis conjecture is single-invocation. Configuration **cvc4+si-r** executes the procedure from Sect. 3 on all benchmarks having conjectures that it can deduce are 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 Sect. 5.2.

6.1 Benchmarks with syntactic restrictions

We evaluated our implementation on 308 benchmarks taken from the 2014 SyGuS competition [2] and the general track of the 2015 competition. The benchmarks are in a new format for specifying syntax-guided synthesis problems [37], which is supported by CVC4's front end. These 308 benchmarks have an associated grammar that specifies syntactic restrictions on the possible solutions.

We ran all of our experiments on the StarExec cluster [48], comparing results from all the synthesis tools that participated in the general track of SyGuS 2015.[14] CVC4 was one of the entrants and won that track. The other entrants were ESOLVER [51] (indicated as **enum** in our tables), a solver based on Stochastic search techniques [44] (indicated as **stoch**), Sketch [46], and Sosy Toast. We used the version of those tools that was submitted to SyGuS 2015 (which are publicly available) except for CVC4, for which we used its most recent version. We partitioned our benchmarks into two sets based on whether their conjecture was single-invocation or not. In total, CVC4 discovered that 228 of the 308 benchmarks could be rewritten into a form that was single-invocation.

Benchmarks with single-invocation synthesis conjectures The results for benchmarks with single-invocation properties are shown in Table 6. Configuration **cvc4+si-r**, which ignores syntactic restrictions, found a solution very quickly for a majority of benchmarks. It terminated successfully for 221 of 228 benchmarks, and in less than a second for 186 of those. Not all solutions found using this method met the syntactic restrictions. Nevertheless, our methods for reconstructing these solutions (as needed) in the prescribed syntax, implemented in configuration **cvc4+si**, succeeded in 133 cases, or 60% of the total. This is 54 more benchmarks than the 79 solved by the next best solver, ESOLVER. In total, **cvc4+si** solved 66 benchmarks that ESOLVER did not, while ESOLVER solved 12 that **cvc4+si** did not. For each of these 12

13 CVC4 is available at http://cvc4.cs.stanford.edu.

14 A detailed summary of our results can be found at http://lara.epfl.ch/w/cvc4-synthesis.

Table 6 Results for single-invocation synthesis conjectures with syntactic restrictions, showing times (in seconds) and number of benchmarks solved by each solver and configuration over 5 benchmark classes with a 1800 s timeout

	array (31)		**bv** (13)		**hd** (44)		**icfp** (50)		**int** (90)		**Total** (228)	
	#	Time	#	Time	#	Time	#	Time	#	Time	#	Time
cvc4+si	**31**	63.0	**6**	0.1	**44**	0.9	0	0.0	**52**	21.6	**133**	85.6
cvc4+si-r	(31)	56.9	(7)	14.0	(44)	0.9	(50)	5426.8	(89)	22.8	(221)	5521.4
cvc4+sg	1	3.4	0	0.0	26	1253.8	**1**	0.5	27	1378.0	55	2635.7
enum	3	15.4	2	28.3	38	197.5	0	0.0	36	2482.0	79	2723.1
stoch	1	0.4	0	0.0	32	655.9	0	0.0	35	702.0	68	1358.3
sketch	9	2993.6	0	0.0	35	2829.7	0	0.0	17	146.4	61	5969.8
stoast	0	0.0	0	0.0	25	2384.3	**1**	1.2	6	36.6	32	2422.1

The number of benchmarks solved by configuration **cvc4+si-r** are in parentheses because its solutions do not necessarily satisfy the given syntactic restrictions

Table 7 Results for parametric benchmarks class encoding the maximum of n integers

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15
cvc4+si	0.01	0.01	0.02	0.03	0.1	0.1	0.3	0.4	0.6	1.2	1.9	2.6	5.9	9.9
cvc4+sg	1.6	–	–	–	–	–	–	–	–	–	–	–	–	–
enum	0.01	1033.0	–	–	–	–	–	–	–	–	–	–	–	–
stoch	0.3	4.4	997.7	–	–	–	–	–	–	–	–	–	–	–
sketch	3.1	284.0	–	–	–	–	–	–	–	–	–	–	–	–
stoast	28.7	–	–	–	–	–	–	–	–	–	–	–	–	–

The columns show the run time for solvers with a 1800 s timeout

benchmarks, CVC4 was actually able to find a solution, but timed out trying to reconstruct it in the prescribed syntax.

The solutions returned by **cvc4+si-r** were often large, having on the 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 was comparable to that of the solutions produced by other solvers. For instance, among the 66 benchmarks solved by both ESOLVER and **cvc4+si**, the former produced a smaller solution in 25 cases. However, only in 4 cases did **cvc4+si** produce a solution that had 20 more subterms than the solution produced by ESOLVER. This indicates that in addition to having a high precision, the techniques from Sect. 5 used for solution reconstruction are generally effective at producing succinct solutions for this benchmark library.[15]

Configuration **cvc4+sg** does not take advantage of the fact that a given synthesis conjecture is single-invocation. However, it was able to solve 55 of those benchmarks. 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. We found the solutions returned by **cvc4+sg** were no larger than those returned by ESOLVER on the 50 benchmarks they both solved. This provides an experimental confirmation that the fairness techniques for term size described in Sect. 4 ensure minimal size solutions.

We observe that for certain classes of benchmarks, configuration **cvc4+si** scales significantly better than state-of-the-art synthesis tools. For instance, for benchmarks from the **array** class,[16] whose solutions are loop-free programs that compute the first instance of an element in a sorted array, **cvc4+si** was able to reconstruct solutions for arrays of size 15 (the largest benchmark in the class) in 0.3 s, and solved each of the benchmarks in the class but 8 within 1 s. In contrast, the next best tool of those shown in Table 8 was Sketch, which solved a problem for an array of length 7 in approximately 2 min, but timed out for larger benchmarks. Similarly, for the parametric class of problems for synthesizing a function that computes the maximum of n integer inputs, **cvc4+si** outperforms the other tools shown here by an order of magnitude or more. Table 7 shows the comparison with other tools on such benchmarks. The Stochastic solver is able to solve the problem for $n = 3$ in approximately 15 min, whereas CVC4 easily scales to $n = 15$ and more.

Benchmarks with non-single-invocation synthesis conjectures The configuration denoted **cvc4+sg** is the only CVC4 configuration that can process benchmarks with synthesis con-

[15] As an exception, for the **array** class of benchmarks, **cvc4+si** found solutions that were rewritten into exponentially larger ones during solution reconstruction to meet the syntactic restrictions.

[16] These benchmarks, as contributed to the SyGuS benchmark set, use integer variables only; they were generated by expanding fixed-size arrays and actually contain no array operations.

Table 8 Results for non-single-invocation synthesis conjectures with syntactic restrictions, showing times (in seconds) and numbers of benchmarks solved over 5 benchmark classes with a 1800 s timeout

	int (8)		**invg** (26)		**invgu** (26)		**MP** (12)		**vctrl** (8)		Total (80)	
	#	Time	#	Time	#	Time	#	Time	#	Time	#	Time
cvc4+sg	2	0.1	20	2601.0	21	908.1	4	925.5	**5**	807.7	52	5242.4
enum	**3**	1.3	**23**	7.8	**23**	9.5	**9**	67.6	4	262.8	**62**	349.0
stoch	**3**	5.2	10	4.6	21	40.7	2	65.4	3	317.3	39	433.2
sketch	**3**	15.6	9	68.1	8	65.5	0	0.0	0	0.0	20	149.2
stoast	2	1.1	0	0.0	14	2805.8	0	0.0	1	21.4	17	2828.2

Table 9 Results for conditional linear arithmetic synthesis conjectures without syntactic restrictions

	cvc4+si-r		**AlchemistCSDT**		**AlchemistCS**	
	#	Time	#	Time	#	Time
Total (73)	**73**	30.1	43	2340.8	33	1460.0

Table shows runtime and number solved for a 1800 s timeout

jectures that are not single-invocation. The results for CVC4 and the other entrants of the 2015 SyGuS competition on the 80 non-single-invocation benchmarks from our set are shown in Table 8. Configuration **cvc4+sg** solved 52 of them, while ESOLVER solved 62. In more detail, ESOLVER solved 12 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 competitive with the state of the art on these benchmarks, solving less than **esolver**, but nevertheless noticeably more than the other solvers of the 2015 competition.

Overall results In total, over the entire SyGuS 2014 benchmark set, 185 benchmarks can be solved by a configuration of CVC4 that, whenever possible, runs the methods for single-invocation properties described in Sect. 3, and otherwise runs the method described in Sect. 4. This number is 44 higher than the 141 benchmarks solved in total by ESOLVER. Running both configuration **cvc4+sg** and **cvc4+si** in parallel[17] solves 193 benchmarks, indicating that CVC4 is highly competitive with state-of-the-art tools for syntax guided synthesis. CVC4's performance is noticeably better than all other solvers on single-invocation properties, where our new quantifier instantiation techniques give it a distinct advantage.

6.2 Benchmarks without syntactic restrictions

We also evaluated our implementation on the 73 benchmarks from the conditional linear integer arithmetic track of the 2015 SyGuS competition that do not have associated syntactic restrictions. We found that 71 of these benchmarks had conjectures that were single-invocation. This track was won by CVC4. Here, we compare its performance against the other entrants of this track, two versions of the Alchemist tool [43].

The results over the 73 benchmarks are shown in Table 9. The most recent version of CVC4 is able to solve every benchmark in this set with a total time of 30.1 s, solving 66 of them within 1 s per benchmark. By contrast, the second best solver configuration, **AlchemistCSDT**, is able to solve only 43 of the 73 benchmarks. This shows that the techniques in CVC4 for handling

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

single invocation properties are able to scale significantly better than other approaches for synthesis for linear arithmetic.

All 43 benchmarks solved by **AlchemistCSDT** were also solved by CVC4. For these 43 benchmarks, the average term size of solutions produced by Alchemist was 116.4, and the average term size of solutions produced by CVC4 was 317.8. For 37 benchmarks, Alchemist produced a solution smaller or the same size as CVC4, and for 13 benchmarks, CVC4 produced a solution smaller or the same size as Alchemist. For 23 benchmarks, the solution produced by CVC4 did not have more than 10 more subterms than the solution produced by Alchemist. Thus, we conclude that CVC4 is able to find solutions much faster than existing approaches, but could benefit from additional techniques to reduce solution size. For instance, such techniques could be run as a post-processor to the solutions that CVC4 produces, which can be large but are produced very quickly. The development of such techniques is the subject of future work.

7 Related work

Early work on synthesis established a connection with automated theorem proving [18,30], including the formulation of certain synthesis problems using ∀∃ formulas. This work used resolution-based techniques. We remarked early on that the capabilities of theorem provers were a bottleneck for effective synthesis.

The work on software synthesis procedures [26–28] introduces a particular notion of synthesis equivalent to decision procedures. Whereas the proposed framework of synthesis procedures is more general, the reported instances of that framework are based on modified quantifier elimination procedures; they were implemented and have theoretical completeness properties but do not have the efficiency and scalability of SMT solvers.

Recent work on synthesis has exploited recent advances in theorem proving, particularly in SAT and SMT solving, which have already proven successful in related domains of hardware and software verification. The traditional strength of SAT/SMT solvers has been in reasoning about non-quantified formulas, so many synthesis methods deal with quantifiers outside of the solver. These approaches typically generate a sequence of increasingly more precise non-quantified queries for the solver, using returned counterexamples to refine the query. This approach is often called Counterexample-Guided Inductive Synthesis (CEGIS). A CEGIS-based tool that pioneered recent interest in synthesis from complex specifications is Sketch [45,45,46], which has been applied in a number of domains and uses a SAT solver. Other approaches use SMT solvers combined with dedicated implementations of search outside of the solver, which can take domain-specific constraints into account, such as synthesis of bit-manipulating programs [22,34]. A typical role of the SMT solver there is to validate candidate solutions and provide counterexamples that guide subsequent search. In these approaches, the SMT solvers thus receive a large number of separate queries, with limited communication between these different steps. Other approaches, such as symbolic term exploration [23,24], also use an SMT solver to explore a representation of the space of solutions, but they too are implemented outside the SMT solver. More recent work has leveraged specialized support for ∃∀ solving in the SMT solver Yices for synthesizing bit-vector manipulation programs [50].

Constructive logic has traditionally recognized the usefulness of explicit witnesses, taking the (arguably extreme) viewpoint of requiring all quantifiers to be backed up by witnesses. The efforts have largely evolved around logical foundations, interactive provers [10], and type systems. For decidable theories such as integer linear arithmetic the constructive requirements are less relevant; what is more important are algorithmic aspects of automated reasoners. Insights from constructive logic are more likely to be relevant for more general synthesis tasks, such as synthesis of recursive and higher-order functions, which fall outside of the current SyGuS format.

Reactive synthesis considers a more general scenario of synthesizing a transition system that potentially interacts with a specified environment in each step of the execution. Often the specifications are given in linear temporal logic [35] or its fragments [9]. The unbounded and infinite execution traces present a source of infinity for such a model, even if the transition system itself is finite. The original techniques for this problem are based on automata theory and appear to have very high lower bounds on complexity. Recent breakthroughs in bounded synthesis, however, leverage once again SMT solvers [15], reducing a number of classes of reactive and distributed synthesis problems to quantified constraints over bounded linear order, providing further evidence that SMT solvers with support for quantifiers are a natural underlying technology for synthesis.

To handle quantified formulas, SMT solvers typically either employ general-purpose heuristic techniques that are incomplete [13,16,32,41] or specialized techniques for quantified formulas in (possibly decidable) background theories. An approach for lazy quantifier elimination for linear real arithmetic was developed in the context of SMT solving in [31], and integrated into the algorithm used by modern SMT solvers in [8]. Approaches for quantifier elimination based on lazy enumeration have been used in the context of SMT-based model checking [25]. Counterexample-guided satisfiability procedures have also been developed for quantified Boolean formulas [20,21] and quantified bit-vector formulas [14,53], although these works do not focus on extraction of witness functions for synthesis conjectures.

8 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 the SMT solver CVC4. This is, to the best of our knowledge, the first implementation of a synthesis procedure inside an SMT solver; moreover, it is highly competitive with dedicated synthesis solvers. In particular, using a novel quantifier instantiation technique and a solution enumeration technique for the theory of algebraic datatypes, our implementation is highly competitive with the state of the art represented by the systems that participated in the 2015 syntax-guided synthesis competition. Furthermore, for the important class of single-invocation problems when syntax restrictions permit the if-then-else operator, our implementation significantly outperforms those systems.

Acknowledgements We would like to thank Liana Hadarean for helpful discussions on the normal form used in CVC4 for bit vector terms, and Tim King for his contributions to the ground linear arithmetic solver in CVC4. Funding was provided by European Research Council (BE) (Grant No. 306484), Schweizerischer Nationalfonds zur Förderung der Wissenschaftlichen Forschung (CH) (Grant No. 200020_159949), Directorate for Computer and Information Science and Engineering (Grant No. CNS 1228768).

References

1. Aloul FA, Ramani A, Markov IL, Sakallah KA (2002) Solving difficult sat instances in the presence of symmetry. In: Proceedings of the 39th annual design automation conference. ACM, pp 731–736
2. Alur R, Bodik R, Dallal E, Fisman D, Garg P, Juniwal G, Kress-Gazit H, Madhusudan P, Martin MMK, Raghothaman M, Saha S, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2014) Syntax-guided synthesis. In: Marktoberdrof NATO proceedings (to appear). http://sygus.seas.upenn.edu/files/sygus_extended.pdf, retrieved 2015-02-06
3. Alur R, Bodík R, Juniwal G, Martin MMK, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2013) Syntax-guided synthesis. In: FMCAD. IEEE, pp 1–17
4. Alur R, Martin MMK, Raghothaman M, Stergiou C, Tripakis S, Udupa A (2014) Synthesizing finite-state protocols from scenarios and requirements. In: Yahav E (ed) Haifa verification conference, LNCS, vol 8855, pp 75–91. Springer. doi:10.1007/978-3-319-13338-6_7
5. Barrett C, Conway C, Deters M, Hadarean L, Jovanovic D, King T, Reynolds A, Tinelli C (2011) CVC4. In: Proceedings of CAV'11, LNCS, vol 6806. Springer, pp 171–177
6. Barrett C, Deters M, de Moura LM, Oliveras A, Stump A (2013) 6 years of SMT-COMP. JAR 50(3):243–277. doi:10.1007/s10817-012-9246-5
7. Barrett C, Shikanian I, Tinelli C (2007) An abstract decision procedure for satisfiability in the theory of inductive data types. J Satisf Boolean Model Comput 3:21–46
8. Bjørner N (2010) Linear quantifier elimination as an abstract decision procedure. In: Giesl J, Hähnle R (eds) IJCAR, LNCS, vol 6173, pp 316–330. Springer. doi:10.1007/978-3-642-14203-1_27
9. Bloem R, Jobstmann B, Piterman N, Pnueli A, Sa'ar Y (2012) Synthesis of reactive(1) designs. J Comput Syst Sci 78(3):911–938. doi:10.1016/j.jcss.2011.08.007
10. Constable RL, Allen SF, Bromley M, Cleaveland R, Cremer JF, Harper RW, Howe DJ, Knoblock TB, Mendler NP, Panangaden P, Sasaki JT, Smith SF (1986) Implementing mathematics with the Nuprl proof development system. Prentice Hall, Englewood Cliffs
11. Cousot P (2005) Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In: Cousot R (ed) VMCAI, LNCS, vol 3385. Springer, pp 1–24. doi:10.1007/978-3-540-30579-8_1
12. Déharbe D, Fontaine P, Merz S, Paleo BW (2011) Exploiting symmetry in SMT problems. In: Automated deduction—CADE-23. Springer, pp 222–236
13. Detlefs D, Nelson G, Saxe, JB (2003) Simplify: a theorem prover for program checking. Technical report. J ACM
14. Dutertre B (2015) Solving exists/forall problems with yices. In: Workshop on satisfiability modulo theories
15. Finkbeiner B, Schewe S (2013) Bounded synthesis. STTT 15(5–6):519–539. doi:10.1007/s10009-012-0228-z
16. Ge Y, Barrett C, Tinelli C (2007) Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning F (ed) CADE, LNCS, vol 4603. Springer, pp 167–182. doi:10.1007/978-3-540-73595-3_12
17. Ge Y, de Moura L (2009) Complete instantiation for quantified formulas in satisfiability modulo theories. In: Proceedings of CAV'09, LNCS, vol 5643. Springer, pp 306–320. doi:10.1007/978-3-642-02658-4_25
18. Green CC (1969) Application of theorem proving to problem solving. In: Walker DE, Norton LM (eds) IJCAI. William Kaufmann, Los Altos, pp 219–240
19. Jacobs S, Kuncak V (2011) Towards complete reasoning about axiomatic specifications. Verification, model checking, and abstract interpretation. Springer, Berlin, pp 278–293
20. Janota M, Klieber W, Marques-Silva J, Clarke E (2012) Solving QBF with counterexample guided refinement. In: International conference on theory and applications of satisfiability testing. Springer Berlin, pp 114–128 (2012)
21. Janota M, Silva JPM (2011) Abstraction-based algorithm for 2qbf. In: Theory and applications of satisfiability testing—SAT 2011—14th international conference, SAT 2011, Proceedings, pp 230–244, Ann Arbor, MI, USA, 19–22 June 2011
22. Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Oracle-guided component-based program synthesis. In: Kramer J, Bishop J, Devanbu PT, Uchitel S (eds) ICSE. ACM, pp 215–224. doi:10.1145/1806799.1806833
23. Kneuss E, Koukoutos M, Kuncak V (2015) Deductive program repair. In: Kroening D, Pasareanu CS (eds) CAV, LNCS, vol 9207. Springer, pp 217–233. doi:10.1007/978-3-319-21668-3_13
24. Kneuss E, Kuraj I, Kuncak V, Suter P (2013) Synthesis modulo recursive functions. In: Hosking AL, Eugster PT, Lopes CV(eds) OOPSLA. ACM, pp 407–426. doi:10.1145/2509136.2509555
25. Komuravelli A, Gurfinkel A, Chaki S (2014) SMT-based model checking for recursive programs. In: Computer aided verification. Springer

26. Kuncak V, Mayer M, Piskac R, Suter P (2010) Complete functional synthesis. In: Zorn BG, Aiken A (eds) PLDI, pp 316–329. ACM. doi:10.1145/1806596.1806632
27. Kuncak V, Mayer M, Piskac R, Suter P (2012) Software synthesis procedures. CACM 55(2):103–111. doi:10.1145/2076450.2076472
28. Kuncak V, Mayer M, Piskac R, Suter P (2013) Functional synthesis for linear arithmetic and sets. STTT 15(5–6):455–474. doi:10.1007/s10009-011-0217-7
29. Madhavan R, Kuncak V (2014) Symbolic resource bound inference for functional programs. In: Biere A, Bloem R (eds) CAV, LNCS, vol 8559. Springer, pp 762–778. doi:10.1007/978-3-319-08867-9_51
30. Manna Z, Waldinger RJ (1980) A deductive approach to program synthesis. TOPLAS 2(1):90–121. doi:10.1145/357084.357090
31. Monniaux D (2010) Quantifier elimination by lazy model enumeration. In: Touili T, Cook B, Jackson P (eds) CAV, LNCS, vol 6174. Springer, pp 585–599. doi:10.1007/978-3-642-14295-6_51
32. de Moura LM, Bjørner N (2007) Efficient e-matching for SMT solvers. In: F. Pfenning (ed) CADE, LNCS, vol 4603. Springer, pp 183–198. doi:10.1007/978-3-540-73595-3_13
33. Nieuwenhuis R, Oliveras A, Tinelli C (2006) Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J ACM 53(6):937–977
34. Perelman D, Gulwani S, Grossman D, Provost P (2010) Test-driven synthesis. In: O'Boyle MFP, Pingali K (eds) PLDI. ACM, p 43. doi:10.1145/2594291.2594297
35. Pnueli A, Rosner R (1989) On the synthesis of a reactive module. In: Conference record of the sixteenth annual ACM symposium on principles of programming languages, pp 179–190, Austin, TX, USA, 11–13 Jan 1989. doi:10.1145/75277.75293
36. Presburger M (1929) Über die Vollständigkeit eines gewissen Systems der Aritmethik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du premier Congrès des Mathématiciens des Pays slaves, Warsawa, pp 92–101
37. Raghothaman M., Udupa A (2014) Language to specify syntax-guided synthesis problems. CoRR arXiv:1405.5590
38. Reynolds A, Deters M, Kuncak V, Tinelli C, Barrett CW (2015) Counterexample-guided quantifier instantiation for synthesis in SMT. In: Computer aided verification—27th international conference, CAV 2015, Proceedings, Part II, pp 198–216, San Francisco, CA, USA, 18–24 July 2015
39. Reynolds A, King T, Kuncak V (2015) An instantiation-based approach for solving quantified linear arithmetic. CoRR arXiv:1510.02642
40. Reynolds A, Tinelli C, Goel A, Krstić S, Deters M, Barrett C (2013) Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina MP (ed) Proceedings of the 24th international conference on automated deduction, Lake Placid, NY, USA, Lecture notes in computer science, vol 7898. Springer, pp 377–391
41. Reynolds A, Tinelli C, Moura LD (2014) Finding conflicting instances of quantified formulas in SMT. In: Formal methods in computer-aided design (FMCAD)
42. Ryzhyk L, Walker A, Keys J, Legg A, Raghunath A, Stumm M, Vij M (2014) User-guided device driver synthesis. In: Flinn J, Levy H (eds) OSDI. USENIX Association, pp 661–676
43. Saha S, Garg P, Madhusudan P (2015) Alchemist: learning guarded affine functions. In: Kroening D, Psreanu CS (eds) Computer aided verification, Lecture notes in computer science, vol 9206, pp 440–446. Springer. doi:10.1007/978-3-319-21690-4_26
44. Schkufza E, Sharma R, Aiken A (2013) Stochastic superoptimization. SIGPLAN Not 48(4):305–316. doi:10.1145/2499368.2451150
45. Solar-Lezama A (2013) Program sketching. STTT 15(5–6):475–495. doi:10.1007/s10009-012-0249-7
46. Solar-Lezama A, Tancau L, Bodík R, Seshia SA, Saraswat VA (2006) Combinatorial sketching for finite programs. In: Shen JP, Martonosi M (eds) ASPLOS. ACM, pp 404–415. doi:10.1145/1168857.1168907
47. Srivastava S, Gulwani S, Foster JS (2013) Template-based program verification and program synthesis. STTT 15(5–6):497–518. doi:10.1007/s10009-012-0223-4
48. Stump A, Sutcliffe G, Tinelli C (2014) 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
49. Svenningsson J, Axelsson E (2012) Combining deep and shallow embedding for EDSL. In: Trends in functional programming—13th international symposium, TFP 2012, Revised selected papers, pp 21–36, St. Andrews, UK, 12–14 June 2012. doi:10.1007/978-3-642-40447-4_2
50. Tiwari A, Gascón A, Dutertre B (2015) Program synthesis using dual interpretation. In: Automated deduction—CADE-25—25th international conference on automated deduction, Proceedings, Berlin, Germany, 1–7 Aug 2015, pp 482–497
51. Udupa A, Raghavan A, Deshmukh JV, Mador-Haim S, Martin MM, Alur R (2013) Transit: specifying protocols with concolic snippets. In: PLDI. ACM, pp 287–296. doi:10.1145/2491956.2462174

52. Wildmoser M, Nipkow T (2004) Certifying machine code safety: shallow versus deep embedding. In: Theorem proving in higher order logics, 17th international conference, TPHOLs 2004, Proceedings, pp 305–320, Park City, UT, USA, 14–17 Sept 2004. doi:10.1007/978-3-540-30142-4_22
53. Wintersteiger CM, Hamadi Y, De Moura L (2013) Efficiently solving quantified bit-vector formulas. Form Methods Syst Des 42(1):3–23