Scalable Algorithms for Abduction via Enumerative Syntax-Guided Synthesis

Andrew Reynolds[1], Haniel Barbosa[2], Daniel Larraz[1], and Cesare Tinelli[1](✉)

[1] Department of Computer Science, The University of Iowa, Iowa City, USA
cesare-tinelli@uiowa.edu

[2] Department of Computer Science, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil

Abstract. The abduction problem in logic asks whether there exists a formula that is consistent with a given set of axioms and, together with these axioms, suffices to entail a given goal. We propose an approach for solving this problem that is based on syntax-guided enumeration. For scalability, we use a novel procedure that incrementally constructs a solution in disjunctive normal form that is built from enumerated formulas. The procedure can be configured to generate progressively weaker and simpler solutions over the course of a run of the procedure. Our approach is fully general and can be applied over any background logic that is handled by the underlying SMT solver in our approach. Our experiments show our approach compares favorably with other tools for abductive reasoning.

1 Introduction

The abduction problem for theory T, a set of axioms A and goal G asks whether there exists a formula φ such that: (i) $\mathsf{A} \wedge \varphi$ is T-satisfiable and (ii) $\mathsf{A} \wedge \varphi \models_T \mathsf{G}$. In other words, it asks for a formula φ that is consistent with the axioms and when added to it allows the goal to be proven. Ideally, φ should be as weak as possible and typically, it is expected to satisfy additional syntactic restrictions, such as, for instance, on its quantifier prefix. Abductive reasoning has gained a variety of applications recently, including extending knowledge bases for failed verification conditions [16] and invariant generation [17,20]. Despite the usefulness of abductive reasoning, and the recent development of a few abductive reasoners, such as GPiD [19] and Explain [15], general tools for automatic abductive inference are not yet mainstream.

Independently from the research on abduction, many high-performance general-purpose solvers for syntax-guided synthesis (SyGuS) have also been developed in the past decade. These solvers have been applied successfully in a number of domains, including the implementation of network protocols [36],

This work was partially supported by NSF grant #1656926 and DARPA grants #N66001-18-C-4006 and #N66001-18-C-4012.

data processing [22], and code optimization [29]. The performance and scalability of SyGuS solvers has made considerable progress in recently years, as demonstrated by an annual competition [4].

In this paper, we investigate scalable approaches to solving the abduction problem using (enumerative) syntax-guided synthesis techniques. We impose no requirements on the background theory T other than it must be supported by an existing SMT solver and amenable to syntax-guided synthesis, as we explain in more detail later. Our immediate goal is to leverage the power of syntax-guided synthesis solvers. Our longer term goal is to standardize the interface for these solvers for abduction problems and make them available to users of program analysis and automated reasoning who would benefit from high performance automated reasoning systems for abduction.

Contributions

- We introduce a novel procedure for solving abduction problems using enumerative syntax-guided synthesis.
- We give an extension of the procedure that is capable of generating progressively weaker solutions to a given abduction problem.
- We provide an implementation of these techniques in cvc4sy [31], a state-of-the-art SyGuS solver implemented within the SMT solver cvc4 [8], and discuss several experiments we designed to test its effectiveness. We show that it has compelling advantages with respect to to other approaches for abduction including those implemented in Explain [15] and GPiD [19].

2 Preliminaries

We work in the context of many-sorted first-order logic with equality ($\simeq$) and assume the reader is familiar with the notions of signature, terms, and so on (see, e.g., [21]). A *theory* is a pair $T = (\Sigma, I)$ where Σ is a signature and I is a non-empty class of Σ-interpretations, the *models of* T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in I only in how it interprets the variables is also in I) and isomorphism. A Σ-formula φ is *T-satisfiable* (respectively, *T-unsatisfiable*) if it is satisfied by some (resp., no) interpretation in I. A satisfying interpretation for φ is a *model* φ. A formula φ is *valid in* T (or *T-valid*), written $\models_T \varphi$, if every model of T is a model of φ. We write $\varphi[\boldsymbol{x}]$ for a tuple $\boldsymbol{x}$ of distinct variables to indicate that the free variables of φ occur in $\boldsymbol{x}$. Given $\varphi[\boldsymbol{x}]$, we write $\varphi[\boldsymbol{t}]$ to denote the result of replacing every occurrence of every variable of $\boldsymbol{x}$ in φ with the corresponding term in the tuple $\boldsymbol{t}$. We write conjunctions of formulas as sets.

Syntax-Guided Synthesis (SyGuS). Syntax-guided synthesis [2] is a recent paradigm for automated synthesis that combines semantic and syntactic restrictions on the space of solutions. Specifically, a SyGuS problem for a function f in a theory T consists of

1. *semantic restrictions*, a specification given by a (second-order) T-formula of the form $\exists f.\, \forall \boldsymbol{x}.\, \varphi[f, \boldsymbol{x}]$, and
2. *syntactic restrictions* on the solutions for f, given by a context-free grammar $\mathcal{R}$.

The grammar $\mathcal{R}$ is a triple (s_0, S, R) where s_0 is an initial symbol, S is a set of symbols with $s_0 \in S$, and R is a set of *production rules* of the form $s \rightarrow t$, where $s \in S$ and t is a term built from the symbols in the signature of theory T, free variables, and symbols from S. The rules define a rewrite relation over such terms, also denoted by $\rightarrow$, as expected. We say a term t is *generated* by $\mathcal{R}$ if $s_0 \rightarrow^* t$ where $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$ and t does not contain symbols from S. For example, the terms x, $(x + x)$ and $((1 + x) + 1)$ are all generated by the grammar $\mathcal{R} = (\mathsf{I}, \{\mathsf{I}\}, \{\mathsf{I} \rightarrow x, \mathsf{I} \rightarrow 1, \mathsf{I} \rightarrow (\mathsf{I} + \mathsf{I})\})$. A *solution for the SyGuS problem for* f is a lambda term $\lambda \boldsymbol{x}.e$ of the same type as f such that (i) $\forall \boldsymbol{x}.\, \varphi[\lambda \boldsymbol{x}.e, \boldsymbol{x}]$ is T-valid and (ii) e is generated by $\mathcal{R}$.

A number of recent approaches for the syntax-guided synthesis problem exist that target specific classes of semantic and syntactic restrictions, including programming-by-examples [22], single invocation conjectures [32], and pointwise specifications [5,27]. General purpose methods for solving the syntax-guided synthesis problem are generally based on *enumerative counterexample-guided inductive synthesis* (CEGIS) [34,35]. Enumerative approach uses a grammar to generate candidate solutions systematically based on some term ordering, typically term size (e.g., the number of non-nullary function applications in the term). The generated candidate solutions are then tested for correctness using a verification oracle (typically an SMT solver). This process is accelerated by the use of *counterexamples* for previously discarded candidates, i.e., valuations for the input variables $\boldsymbol{x}$, or *points*, that witness the failure of those candidates to satisfy the specification. Despite its simplicity, enumerative CEGIS is the de-facto approach for solving the general class of SyGuS problems, as implemented in a several recent tools, notably cvc4sy [31] and the enumerative solver ESolver [3]. Its main downside remains scalability to cases where the required solution is very large. As we will show in Sect. 4, we present a more scalable procedure for the abduction problem that builds on top of enumerative CEGIS and is capable of quickly finding (conjunctive) solutions.

3 The Abduction Problem

In general, the abduction problem for a set A of axioms and a goal G is the problem of finding a formula S that is consistent with A and, together with A, entails the goal. We refine the problem by restricting it to first-order logic and to a given background theory T, and also considering syntactic restrictions on the solution S. We refer to this as the *syntax-restricted abduction problem*, which we formalize in the following definition.

Definition 1 (Abduction Problem). *The (syntax-restricted) abduction problem for a theory T, a conjunction $\mathsf{A}[\boldsymbol{x}]$ of axioms, a goal $\mathsf{G}[\boldsymbol{x}]$ and a grammar*

R*, where axioms and goal are first-order formulas, is that of finding a first-order formula* $\mathsf{S}[\boldsymbol{x}]$ *such that:*

1. $\mathsf{A} \wedge \mathsf{S} \models_T \mathsf{G}$*,*
2. $\mathsf{A} \wedge \mathsf{S}$ *is T-satisfiable, and*
3. S *is generated by grammar* R*.*

In practice, as in SyGuS, syntactic restrictions on the solution space may be used to capture user-requirements on the desired shape of a solution. They can also be used as a mechanism for narrowing the search space to one where one believes the solver is likely to find a solution. Observe that the formulation of the problem includes the case with no syntactic restriction as a trivial case of a grammar that accepts all formulas in the signature of the theory T. In the abduction solver we have developed for this work, the syntax restriction is optional. When it is missing, a grammar generating the full language is constructed internally automatically.

Syntax-restricted abduction bears a strong similarity to SyGuS.[1] We exploit this similarity by leveraging much of the technology we developed for SyGuS, with the goal of achieving generality and scalability.

Normally, an abduction problem admits many solutions. Thus, it may be useful to look for solutions that optimize certain criteria, such as generality with respect to entailment in T, or minimality with respect to size or number of free variables. Our evaluation contains several case studies where we explore this aspect in detail.

Recent Applications. Abduction has a long history in logic and automatic reasoning (see, e.g., [24]). More recently, it has found many useful applications in program analysis. It has been used for identifying the possible facts a verification tool is missing to either discharge or validate a verification condition [16], inferring library specifications that are needed for verifying a client program [37], and synthesizing specifications for multiple unknown procedures called from a main program [1]. Other applications of abduction includes loop invariant generation [17,20], where it is used to iteratively strengthen candidate solutions until they are inductive and strong enough to verify a program, and compositional program verification [25], where it is used for inferring not only loop invariants but also preconditions required for the invariants to hold. Abductive inference has also been applied to modular heap reasoning [12], and the synthesis of missing guards for memory safety [18].

4 Abduction via Enumerative Syntax-Guided Synthesis

In this section, we fix a theory T and describe our approach for solving the abduction problem in T using enumerative syntax-guided synthesis. We first present a basic procedure for abduction in the following section, and then extend it to generate (conjunctive) solutions in a highly scalable manner. We then describe how

[1] In fact, it could be readily recast as SyGuS, if one ignored Condition 2 in Definition 1.

```
GetAbductBasic(axioms $\mathsf{A}[\boldsymbol{x}]$, goal $\mathsf{G}[\boldsymbol{x}]$, grammar $\mathsf{R}$)
 1: Let $\mathsf{P} = \varnothing$    // set of points
 2: loop
 3:    Let $c[\boldsymbol{x}] = \mathsf{NextEnum}(\mathsf{R})$
 4:    if $\mathsf{Eval}(c, \boldsymbol{p}) = \bot$ for all $\boldsymbol{p} \in \mathsf{P}$ then
 5:       if $c \wedge \mathsf{A} \wedge \neg\mathsf{G}$ is $T$-satisfiable then
 6:          $\mathsf{P} := \mathsf{P} \cup \{\boldsymbol{p}\}$ with $\boldsymbol{p}$ such that $\mathsf{Eval}(c \wedge \mathsf{A} \wedge \neg\mathsf{G}, \boldsymbol{p}) = \top$
 7:       else if $c \wedge \mathsf{A}$ is $T$-satisfiable then
 8:          return $c$
 9:       end if
10:    end if
11: end loop
```

Fig. 1. Basic procedure for the abduction problem for axioms A, goal G and grammar R.

either approach can be extended to be incremental so that it constructs progressively logically weaker solutions over time. For simplicity, *we restrict ourselves to abduction problems where axioms, goals, and solutions are quantifier-free.* Note, however, that the procedure can be used for abduction problems where these components are quantified, as long the restrictions below (lifted to quantified formulas) are satisfied.

Requirements on *T*. We assume that the T-satisfiability of quantifier-free formulas is decidable. For each sort of T, we also assume a distinguished set of variable-free terms of that sort which we call *values* (e.g., numerals and negated numerals in the case of integer arithmetic) such that every T-satisfiable formula is satisfied by a valuation of its free variables to sort elements denoted by values. Finally, we require the availability of a computable function Eval that takes a first-order formula $\varphi[\boldsymbol{x}]$ and a tuple $\boldsymbol{p}$ of values of the same length as $\boldsymbol{x}$, and returns $\top$ if $\varphi[\boldsymbol{p}]$ is T-satisfiable and $\bot$ otherwise. These restrictions are met by most theories used in Satisfiability Modulo Theories (SMT).

4.1 Enumerative Counterexample-Guided Inductive Synthesis for Abduction

We start with a basic CEGIS-style synthesis procedure for solving the syntax-restriction abduction problem where points that represent counterexamples for candidate solutions are cached and used to discard subsequent candidates. The procedure is presented in Fig. 1. It takes as input: axioms A, goal G and grammar R, and maintains an internally set P of points that satisfy the axioms and falsify the goal. On line 3, it invokes the stateful sub-procedure NextEnum(R) which enumerates the formulas generated by grammar R based on enumerative techniques used in SyGuS solvers. We will refer to the return formula c as the current *candidate solution*. Then, using the (fast) evaluation function Eval, it checks at line 4 that c is satisfied by none of the counterexample points in P. If the check fails, the procedure discards c and loops back to line 3 because adding

c to A would definitely be not enough to entail G. If the check succeeds, it also checks, at line 5, whether $c \wedge \mathsf{A} \wedge \neg\mathsf{G}$ is T-satisfiable. If so, it obtains a witness point $\boldsymbol{p}$ for the satisfiability, adds it to current set of points P on line 6, and discards c; otherwise, it checks that c is consistent with A before returning it as a possible solution.

Example 1. Let T be the theory of linear integer arithmetic with the usual signature. Let A be the set $\{y \geqslant 0\}$, let G be the set $\{x+y+z \geqslant 0\}$, and assume R is a grammar generating all linear arithmetic atomic formulas over the variables x, y, z. The results of the procedure are summarized in the table below. We provide, for each iteration, the candidate c generated by syntax-guided enumeration on line 3, the Boolean value of the conditions on lines 4, 5 and 7 of the procedure when applicable, and the point (x, y, z) added to P in when the condition on line 5 evaluates to true. The last column specifies the solution returned on that iteration if any.

#	c	line 4	line 5	$p \in \mathsf{P}$	line 7	return
1	$x \geqslant 0$	true	true	$(0, 0, -1)$		
2	$x < 0$	true	true	$(-1, 0, 0)$		
3	$y \geqslant 0$	false				
4	$y < 0$	true	false		false	
5	$z \geqslant 0$	false				
6	$z < 0$	false				
7	$x+y \geqslant 0$	false				
8	$x+y < 0$	false				
9	$x+z \geqslant 0$	true	false		true	$x+z \geqslant 0$

On the first iteration, the syntax-guided enumeration generates the formula $x \geqslant 0$ as the candidate solution c. This fails to imply the goal, specifically, with $(x, y, z) = (0, 0, -1)$ the axioms and c are satisfied and the goal is falsified. The second candidate fails for similar reasons for point $(-1, 0, 0)$. The check on line 4 fails for five of the next six candidates, with the exception of the candidate $y < 0$. This candidate is falsified by both points in P but it must be discarded since it is inconsistent with the axioms (line 7). Finally, the candidate $x + z \geqslant 0$ generated on the ninth iteration passes all the tests and is returned as a solution for this abduction problem. □

4.2 A Procedure for Abduction Based on Unsat Core Learning

This section extends the procedure from Fig. 1 with techniques that make it scalable when the intended solution to the abduction problem is a conjunction of formulas. The procedure is applicable to cases where the language generated by grammar R is closed under conjunction. In essence, the procedure in this section applies when $s_0 \rightarrow s_0 \wedge s_0$ is a production rule in R where s_0 is the start symbol of R. However, it avoids enumerating conjunctive formulas directly, preferring instead to generate them as sets of (non-conjunctive) formulas.

```
GetAbductUCL(axioms A, goal G, grammar R)
 1: Let $\mathsf{E}, \mathsf{P}, \mathsf{U} = \varnothing$
 2: loop
 3:     $\mathsf{E} := \mathsf{E} \cup \{\mathsf{NextEnum}(\mathsf{R})\}$
 4:     Let $\mathsf{C} = \varnothing$
 5:     while EnsureCexFalsify(C, E, P, U) do
 6:         if $\mathsf{C} \wedge \mathsf{A} \wedge \neg \mathsf{G}$ is $T$-unsatisfiable then
 7:             Let $\mathsf{C}_{min} \subseteq \mathsf{C}$ such that $\mathsf{C}_{min} \wedge \mathsf{A} \wedge \neg \mathsf{G}$ is $T$-unsatisfiable
 8:             if $\mathsf{C}_{min} \wedge \mathsf{A}$ is $T$-satisfiable then
 9:                 return $\mathsf{C}_{min}$
10:             else
11:                 Let $u \subseteq \mathsf{C}_{min}$ such that $u \wedge \mathsf{A}$ is $T$-unsatisfiable
12:                 $\mathsf{U} := \mathsf{U} \cup \{u\}$; $\mathsf{C} := \mathsf{C} \setminus \{e\}$ for some $e \in u$
13:             end if
14:         else
15:             $\mathsf{P} := \mathsf{P} \cup \{p\}$ where $\mathsf{Eval}((\mathsf{C} \wedge \mathsf{A} \wedge \neg \mathsf{G}), p) = \top$
16:         end if
17:     end while
18: end loop

EnsureCexFalsify(candidate C, formulas E, points P, cores U)
 1: while $\mathsf{C} = \varnothing$ or $\mathsf{Eval}(\mathsf{C}, \boldsymbol{p}) = \top$ for some $\boldsymbol{p} \in \mathsf{P}$ do
 2:     if $\mathsf{Eval}(e, \boldsymbol{p}) = \bot$ for some $e \in \mathsf{E}$ and $u \nsubseteq \mathsf{C} \cup \{e\}$ for all $u \in \mathsf{U}$ then
 3:         $\mathsf{C} := \mathsf{C} \cup \{e\}$
 4:     else
 5:         return false
 6:     end if
 7: end while
 8: return true
```

Fig. 2. Procedure for the abduction problem for A, G and R based on unsat core learning.

This procedure is presented in Fig. 2. Similarly to the basic procedure from the previous section, it maintains a set of points P that satisfy the axioms and falsify the goal. Additionally, the new procedure maintains a set E of enumerated formulas, and a set U of subsets of E that are inconsistent with the axioms. The procedure modifies to each of these three sets during the course of its run. Each loop iteration attempts to construct a set C of formulas whose conjunction is a solution to the abduction problem. This is in contrast to the basic procedure from Fig. 1 which considers only individual formulas as candidate solutions.

To construct the candidate set C, the procedure uses a helper function EnsureCexFalsify which ensures that (i) C is non-empty, (ii) the conjunction of the formulas in C is falsified by each point in P and (iii) no subset of C occurs in U. The first condition is to ensure that the candidate is generated by the grammar. The second condition ensures that C along with our axioms suffices to prove the goal. The third condition ensures that C is consistent with the axioms. If we are able to successfully construct a candidate solution set C, then line 6 checks whether that candidate indeed suffices when added to the axioms to show the goal. If it does not, we add a counterexample point to P; otherwise, we construct a (ideally minimal) subset of C_{min} of C that also suffices to show the goal. This information can be readily computed by an SMT solver [10] with support respectively for model generation and for unsatisfiable core generation [13], two features common to most modern solvers, including cvc4. We then check whether C_{min} is consistent with our axioms. If it is consistent, we return it as a solution to the abduction problem; if it is not, we add some subset of it to U that is also inconsistent with the axioms, where again the subset can be computed by an SMT solver with support for unsatisfiable cores. Adding such subset amounts to learning that subset should never be included in future candidate solutions. To maintain the invariant that no subset of C occurs in U, we remove one enumerated formula $e \in u$ from C on line 12. In the case where a point is added to P (line 15) or when an unsat core is added to U (line 12), we run the method $\mathsf{EnsureCexFalsify}$ starting from the current resultant set C. This will force the procedure to try to construct a new candidate solution based on the set E. When this strategy fails to construct a candidate, the inner loop terminates and the next formula is added to E based on syntax-guided enumeration.

We now revisit Example 1. As demonstrated in this example, $\mathsf{GetAbductUCL}$ is often capable of generating solutions to the abduction problem faster than the one from Fig. 1, albeit those solutions may be logically stronger.

Example 2. We revisit Example 1, where A is the set $\{y \geqslant 0\}$ and G is $\{x+y+z \geqslant 0\}$. A run of the procedure from Fig. 2 is summarized in the table below. We list iterations of the outer loop of the procedure (lines 2–18) in the first column of this table. For each iteration, we provide the formula that is added to our pool E (line 3), and the considered candidate set C upon a successful call to $\mathsf{EnsureCexFalsify}$. Notice that the inner loop of the procedure may consider multiple candidates C for a single iteration of the outer loop. For each candidate, when applicable, we give the result of the evaluation of the condition on line 6, the point p added to P if that condition is false (line 15), the minimal candidate set C_{min} constructed on line 7, the evaluation of the condition on line 8, the set of formulas added to our set of unsatisfiable cores if that condition is false (line 12), and finally the formula (if any) returned as a solution (line 9).

#	$e \in \mathsf{E}$	C	line 6	$p \in \mathsf{P}$	C_{min}	line 8	$u \in \mathsf{U}$	return
1	$x \geqslant 0$	$\{x \geqslant 0\}$	false	$(0, 0, -1)$				
2	$x < 0$	$\{x < 0\}$	false	$(-1, 0, 0)$				
		$\{x < 0, x \geqslant 0\}$	true		C	false	$\{x < 0, x \geqslant 0\}$	
3	$y \geqslant 0$							
4	$y < 0$	$\{y < 0\}$	true		C	false	$\{y < 0\}$	
5	$z \geqslant 0$	$\{x \geqslant 0, z \geqslant 0\}$	true		C	true		$x \geqslant 0 \wedge z \geqslant 0$

We assume the same ordered list of formulas enumerated from Fig. 1. On the first iteration, we add $x \geqslant 0$ to our pool of enumerated formulas E. The helper function $\mathsf{EnsureCexFalsify}$ constructs the set $\mathsf{C} = \{x \geqslant 0\}$ since (vacuously) it is true for all points in P. Similar to the first iteration of Fig. 1, on line 6 we learn that $x \geqslant 0$ does not suffice with our axioms to show the goal; a counterexample point is $(x, y, z) = (0, 0, -1)$ which is added to P. Afterwards, $\mathsf{EnsureCexFalsify}$ is not capable of constructing another C since there are no other formulas in E. In contrast to Fig. 1 which discards the formula $x \geqslant 0$ at this point, here it remains in E and can be added as part of C in future iterations.

On the second iteration, we add $x < 0$ to our pool. We check the candidate set $\mathsf{C} = \{x < 0\}$, which fails to imply the goal for counterexample point $(x, y, z) = (-1, 0, 0)$. To construct the next candidate set C, we must find an additional formula from E that evaluates to false on this point (or otherwise we again would fail to imply our goal). Indeed, $x \geqslant 0 \in \mathsf{E}$ evaluates to false on this point, and thus $\mathsf{EnsureCexFalsify}$ returns the set $\{x < 0, x \geqslant 0\}$. This set suffices to prove the goal given the axioms, that is, the condition on line 6 succeeds; the unsatisfiable core C_{min} computed for this query is the same as C. However, on line 8, we learn that this set is inconsistent with our axioms (in fact, the set by itself is equivalent to false). On line 12, we add $\{x < 0, x \geqslant 0\}$ to U. In other words, we learn that *any* solution that contains both these formulas is inconsistent with our axioms. Learning this subset will help prune later candidate solutions. The procedure on this iteration proceeds by removing one of these formulas from our candidate solution set C. Subsequently the helper function $\mathsf{EnsureCexFalsify}$ cannot construct a new candidate subset due to $\{x < 0, x \geqslant 0\} \in \mathsf{U}$ and since no other formulas occur in E.

On the third iteration, $y \geqslant 0$ is added to our pool. However, no candidate solution can be constructed, where notice that $y \geqslant 0$ evaluates to $\top$ on both points in P. On the fourth iteration, $y < 0$ is added to our pool and the candidate solution set $\{y < 0\}$ is constructed, where notice that this formula evaluates to $\perp$ on both points in P. This formula suffices to show the goal from the axioms, but is however inconsistent with our axioms. Thus, $\{y < 0\}$ is added to our set of unsatisfiable cores U. In other words, we have learned that no solution C should include the formula $y < 0$ since it is alone inconsistent with our axioms.

On the fifth iteration, $z \geqslant 0$ is added to our pool. The only viable candidate that falsifies all points in P and does not contain a subset from U is $\{x \geqslant 0, z \geqslant 0\}$. This set is a solution to the abduction problem and so the formula $x \geqslant 0 \wedge z \geqslant 0$ is returned. Due to our assumption that R admits conjunctions, this formula meets the syntax restrictions of our grammar. A run of this procedure required the enumeration of only 5 formulas before finding a solution whereas the basic one in Fig. 1 required 9. $\square$

While the solution in the previous example $x \geqslant 0 \wedge z \geqslant 0$ was found in fewer iterations, notice that it is logically stronger than the solution $x + z \geqslant 0$ produced in Example 1, since $x \geqslant 0 \wedge z \geqslant 0$ entails $x + z \geqslant 0$ but not vice versa. We remark that the main advantage of procedure Fig. 2 is that is typically capable of generating *any* feasible solution to the abduction problem faster than the procedure from Fig. 1. This is especially the case if the only solutions to the abduction problem consist of a large conjunction of literals of small term size $\ell_1 \wedge \ldots \wedge \ell_n$. The basic procedure does not scale to this case, if its enumeration is by formula size, since it will have to wait until the conjunction above is enumerated as an individual formula.

Furthermore, we remark that procedure in this section can be configured to have the same solution completeness guarantees as the basic procedure from Fig. 1. In particular, our choice of e in the EnsureCexFalsify method chooses the most recently enumerated formula when the candidate pool C is empty. Since a single loop of the procedure is terminating and due to the above policy for selection, the procedure will terminate in the worst case when the enumerated pool E contains a formula that by itself is the solution to the synthesis conjecture.

4.3 Incremental Weakening for Abduction

The user may be interested in obtaining an abduction problem solution that maximizes some criteria and is not necessarily the first one discovered by (either of) the procedures we have described so far. In this section, we describe an extension to our approach for abduction that maintains the advantage of returning solutions quickly while still seeking to generate the best solution in the long run according to metric such as logical weakness.

We observe that it is straightforward to extend our enumerative syntax-guided approach to generate *multiple* solutions. We are interested, however, in generating increasingly better solutions over time. We briefly give an overview of how the procedures of Fig. 1 and Fig. 2 can be extended in this way and discuss a few relevant details of the extension. We focus on the problem of generating the *logically weakest* solution to the abduction problem in this section.

Figure 3 presents an incremental procedure for generating (multiple) solutions to a given abduction problem. The procedure requires that the language restriction R admit disjunctive formulas which is the case, for instance, if $s_0 \rightarrow s_0 \vee s_0$ is a production rule in R where s_0 is again the start symbol. It maintains a formula S that, when not $\perp$, represents the logically weakest solution to the abduction problem known so far. In its main loop, on line 3, the procedure calls one of the previous procedures for generating single solutions to the abduction problem (written $\mathsf{GetAbduct}^*$). Line 4 then checks whether a new solution can be constructed that is logically weaker with respect to the axioms than the current one. In particular, this is the case if $\mathsf{C} \wedge \mathsf{A} \wedge \neg \mathsf{S}$ is T-satisfiable, which means that there is at least one point that satisfies the current candidate but not the current solution S. In that case, the current solution S is updated to $\mathsf{S} \vee \mathsf{C}$, which is by construction guaranteed to also be a solution to the abduction problem. If no such point can be found, then C is redundant with respect to the current candidate solution since it does not generalize it. Optionally, the procedure may learn a subset u of C that is also redundant with respect to the current candidate solution. This subset can be learned as an unsatisfiable core when using the procedure GetAbductUCL as the sub-procedure on line 3.

```
GetAbductInc(axioms A, goal G, grammar R)
 1: Let S = ⊥
 2: loop
 3:     Let C = GetAbduct*(A, G, R).
 4:     if C ∧ A ∧ ¬S is $T$-satisfiable then
 5:         S := S ∨ C
 6:         print Weakest solution so far is S
 7:     else
 8:         // Exclude $\{u\}$ for some $u \subseteq$ C such that $u$ ∧ A ∧ ¬S is $T$-unsatisfiable
 9:     end if
10: end loop
```

Fig. 3. Incremental abduction procedure for axioms A, goal G and grammar R.

4.4 Implementation Details

We implemented the procedures above in the state-of-the-art SMT solver CVC4 [8]. CVC4 incorporates a SyGuS solver, CVC4SY, implementing several strategies for enumerative syntax-guided synthesis [31]. It accepts as input both SMT problems written in the SMT-LIB version 2.6 format [9], and synthesis problems written in the SyGuS version 2.0 format [30]. SMT-LIB version 2.6 is a scripting language that allows one to assert a formula F to the solver with a command of the form (assert F). The solver checks the satisfiability the formulas asserted so far in response to the command (check-sat). We extended CVC4's SMT-LIB parser to support also commands of the form (get-abduct p G R) where p is a symbol, the identifier of the expected solution formula; G is a formula, the goal of the abduction problem; and the optional R is a grammar expressed in the SyGuS version 2.0 format. This command asks the solver to find a formula that is a solution to the abduction problem (A, G), where A, standing for the set of axioms, consists of the conjunction of the currently asserted formulas. The expected response from the solver is a definition of the form (define-fun p () Bool S) where p is the identifier provided in the first argument of get-abduct and S is a formula that solves the abduction problem.

Internally, invoking a get-abduct command causes a synthesis conjecture to be constructed and passed to CVC4SY. The latter normally accepts conjectures of the form $\exists f. \forall \boldsymbol{x}. \varphi[f, \boldsymbol{x}]$ where φ is quantifier-free. Thus, we must pass the abduction problem in two parts: (i) the *synthesis conjecture* $\exists P. \forall \boldsymbol{x}. \neg(P(\boldsymbol{x}) \wedge A \wedge \neg G)$ where $\boldsymbol{x}$ collects the free variables of A and of G,[2] stating that the expected solution P along with the axioms A must entail the goal G, and (ii) a *side condition* $\exists \boldsymbol{x}. P(\boldsymbol{x}) \wedge A$ stating that P must be consistent with the axioms. The synthesis conjecture is of a form that can be readily handled by CVC4SY and processed using its current techniques. We have modified it so that it considers the side condition as well during solving, as described in Figs. 1 and 2.

[2] We assume that all free symbols in A and G are variables.

The procedure in Fig. 2 is implemented as a strategy on top of the basic enumerative CEGIS loop of cvc4sy. We give some noteworthy implementation details here. Firstly, we use a data structure for efficiently checking whether any subset of C occurs in our set of unsatisfiable cores U, which keeps the sets in U in an index and is traversed dynamically as formulas are added to C. We chose enumerated formulas on line 2 of EnsureCexFalsify by selecting first the most recently generated formula, and then a random one amongst those that meet the criteria to be included in C. Finally, since the number of candidate solutions can be exponential in the worst case for a given iteration of the inner loop of this procedure, we use a heuristic where formulas cannot be added to C more than once in the same iteration of the loop, making the number of candidate sets tried on a given iteration linear in the size of E in the worst case.

5 Evaluation

We evaluated our approach[3] in comparison with cvc4sy's enumerative CEGIS, a general purpose synthesis approach, as well as with GPiD [19] and Explain [15], state-of-the-art solvers for similar abduction problems as the one defined here. In the comparison below, we refer to the basic procedure from Fig. 1 as cvc4sy+b and the one from Fig. 2 as cvc4sy+u. Experiments ran on a cluster with Intel E5-2637 v4 CPUs, Ubuntu 16.04. Each execution of a solver on a benchmark was provisioned one core, 300 s and 8 GB RAM.

5.1 Benchmarks

Since abduction tools are generally focused on specific application domains, there is no standard language or benchmark library for evaluation. Moreover, these tools use abduction as part of a larger verification toolchain. As here we did not target a specific application but rather the abduction problem as a whole, an evaluation with their benchmarks would require integrating our solver in the tools as an alternative abduction engine. This was not feasible due to either the source code not being available or the verification and abduction engines being too tightly coupled for us to use our solver as an alternative. Thus we had to generate our own abduction benchmark sets. We did so using benchmarks relevant for verification from SMT-LIB [9], the standard test suite for SMT solvers. We chose as a basis the SMT-LIB logics QF_LIA, QF_NIA, and QF_SLIA due to their relevance for verification. For QF_NIA, we focus on the benchmark family VeryMax and on kaluza for QF_SLIA. In QF_LIA we excluded benchmark families whose benchmarks explode in size without the let operator. This was necessary to allow a comparison with Explain, whose parser does not fully support let, on let-free benchmarks. We considered both benchmarks that were (annotated as) satisfiable and unsatisfiable for generating abduction problems, according to the following methodology.

[3] Full material at http://cvc4.cs.stanford.edu/papers/abduction-sygus/.

Given a *satisfiable* SMT-LIB problem $\varphi = \psi_1 \wedge \cdots \wedge \psi_n$[4] in the theory T, we see it as an encoding of a validity problem $\psi_1 \wedge \cdots \wedge \psi_{n-1} \models \neg\psi_n$ that could not be proven. We consider the abduction problem where G is $\neg\psi_n$, A is $\psi_1 \wedge \cdots \wedge \psi_{n-1}$, and R is a grammar that generates any quantifier-free formula in the language of T over the free variables of G and A. A solution S to this problem allows the validity of φ to be proven, since $\varphi \wedge \mathsf{S}$ is unsatisfiable.

Given an *unsatisfiable* SMT-LIB problem φ, let $U = \{\psi_1, \ldots, \psi_n\}$ be a *minimal* unsatisfiable core for this formula, i.e. any conjunctive set $U \setminus \{\psi\}$, for some $\psi \in U$, is satisfiable. Let $\psi_{\max}$ be U's component with maximal size. We will call $\psi_{\max}$ the *reference* to the abduction problem. We consider the abduction problem whose G is $\neg\psi_{\mathsf{G}}$, for some $\psi_{\mathsf{G}} \in U$ and $\psi_{\mathsf{G}} \neq \psi_{\max}$, whose axioms A are $U \setminus \{\psi_{\mathsf{G}}, \psi_{\max}\}$ and R as before is a grammar that generates any formula in the language of T over the free variables of G and A. A solution S to this problem allows proving the validity of $U \setminus \{\psi_{\mathsf{G}}, \psi_{\max}\} \models \psi_{\mathsf{G}}$, since $U \setminus \{\psi_{\max}\} \cup \{\mathsf{S}\}$ is unsatisfiable. Solving this abduction problem amounts to "completing" the original unsatisfiable core with the further restriction that this completion is at least as weak as the reference, as well as consistent with all but one of the other core components, seen as axioms for the abduction problem.

From satisfiable SMT-LIB benchmarks we generated 2025 abduction problems in QF_LIA, 12214 in QF_NIA and 11954 in QF_SLIA. For unsatisfiable benchmarks we were limited not only by the benchmark annotations but also by being able to find minimal unsatisfiable cores. We used the Z3 SMT solver [14] to generate minimal unsatisfiable cores with a 120s timeout. Excluding benchmarks whose cores had less than three assertions (so we could have axioms, a goal and a reference), we ended up with 97 problems in QF_LIA, 781 in QF_NIA and 2546 in QF_SLIA. We chose the reference as the component of the unsatisfiable core with maximal size and the goal as the last formula in the core (viewed as a list) after the reference was removed.

Table 1. Comparison of abduction problems from originally SAT SMT-LIB benchmarks.

Logic	#	CVC4SY+B			CVC4SY+U		
		Solved	Unique	Weaker	Solved	Unique	Weaker
QF_LIA	2025	721	261	183	594	134	2
QF_SLIA	11954	10902	3	466	10980	81	0
QF_NIA	12214	1492	171	671	1712	391	45
Total	26593	13329	435	1320	13628	606	47

5.2 Finding Missing Assumptions in SAT Benchmarks

In this section we evaluate how effective CVC4SY+B and CVC4SY+U are in (i) finding any solution to the abduction problem and (ii) finding logically weak

[4] SMT-LIB problems are represented as sequences of assertions. Here we considered each ψ_i as one of these assertions.

solutions. The evaluation is done on the abduction problems produced from satisfiable SMT-LIB benchmarks as above. Results are summarized in Table 1. The number of solved problems corresponds to the problems for which a given cvc4 configuration could find a solution within 300s. cvc4sy+u solves a significant number of problems more than cvc4sy+b in all logics but QF_LIA. In both QF_LIA and QF_NIA we can see a significant orthogonality between both approaches. We attribute these both to the fragility of integer arithmetic reasoning, where the underlying ground solver checking the consistency of candidate solutions is greatly impacted by the shape of the problems it is given. Overall, the procedure in cvc4sy+u leads to a better success rate than the basic procedure in cvc4sy+b. Solution strength was evaluated on commonly solved problems considering the solutions produced according to the incremental procedures shown in Sect. 4.3, in which the overall solution is a disjunction of individual solutions found over time. As expected, cvc4sy+u is able to solve more problems but at the cost of often producing stronger (and bigger) solutions than cvc4sy+b. This is particularly the case in QF_SLIA and QF_NIA, in which cvc4sy+u both solves many more problems and often finds stronger solutions.

5.3 Completing UNSAT Cores

Here we evaluate how effective cvc4sy+b and cvc4sy+u are in solving the abduction problem with the extra restriction of finding a solution that is at least as weak as a given reference formula. We use the abduction problems produced from unsatisfiable SMT-LIB benchmarks following the methodology of Sect. 5.1 as the basis for this evaluation.

Table 2. Comparison of abduction problems from originally UNSAT SMT-LIB benchmarks.

Logic	#	cvc4sy+b		cvc4sy+u	
		Solved	Unique	Solved	Unique
QF_LIA	97	6	0	6	0
QF_SLIA	2546	2546	32	2514	0
QF_NIA	781	86	49	41	4
Total	3424	2638	81	2561	4

The results are summarized in Table 2. cvc4sy+b significantly outperforms cvc4sy+u in QF_SLIA, in which the references are very simple formulas (generally with size below 3), for which the specialized procedure of cvc4sy+u is not necessary. Overall, as in the previous section when checking who finds the weakest solution, cvc4sy+b has as advantage over cvc4sy+u for finding solutions as weak as the reference.

5.4 Comparison with Explain

Explain [15] is a tool for abductive inference based on quantifier elimination. It accepts as input a subset of SMT-LIB and we extended it to support abduction problems as generated in Sect. 5.1. However, Explain imposes more restrictions to their solutions, only producing those with a *minimal* number of variables and for which every other solution with those variables is not stronger than it. Their rationale is finding "simple" solutions, according to the above criteria, which are more interesting to their applications. Since we do not apply these restrictions in cvc4, nor is in the scope of this paper incorporating them into our procedure, it should be noted that comparing cvc4 and Explain puts the latter at a disadvantage. We considered satisfiable SMT-LIB problems in the QF_LIA logic for our evaluation, as QF_LIA is better supported by Explain (Table 3).

Table 3. Comparison with Explain in 2025 abduction problems in QF_LIA

	Solved	Unique	Total time
cvc4sy+b	721	261	418849 s
cvc4sy+u	594	125	449424 s
Explain	33	0	532839 s

All problems solved by Explain are solved by cvc4sy+u. Of these 33 problems, cvc4sy+u, in incremental mode, finds a solution with the same minimal number of variables as Explain for 25 of them. Of the 8 problems to which it only finds solutions with more variables, in 4 of them the difference is of a single variable. All other 4 are in the slacks benchmark family, which contains crafted problems. A similar comparison occurs with cvc4sy+b. This shows that even though cvc4 is not optimized to minimize the number of variables it its solutions, it can still often finds solutions that are optimal (or close to optimal) according to Explain's criteria, while solving a much larger number of problems with a fully general approach.

5.5 Comparison with GPiD

We also compared cvc4 with GPiD [19], a framework for generating implicates, i.e. logical consequences of formulas. As Echenim et al. say in their paper, negating the implicate of a satisfiable formula φ yields the "missing hypothesis" for making φ unsatisfiable. Therefore GPiD solves a similar problem to that of Sect. 5.2, differing by they always considering an empty set of axioms and the whole original formula as the goal. Given this similarity, we compare the performance of GPiD in generating implicates for satisfiable benchmarks and of cvc4sy+b and cvc4sy+u in solving abduction problems generated from those same benchmarks. We did not consider the benchmarks from the previous sections because we were not able to produce *abduces*, which are the syntactic components GPiD uses to find implicates, for other logics using the tools in GPiD public repository[5]. Thus we restricted our analysis to 400 abduction problems produced, as per the methodology of Sect. 5.1, from satisfiable QF_UFLIA benchmarks that were used in [19]. Note however that the cvc4 configurations will require solutions to be consistent with all but the last assertion in the problems (which are the axioms in the respective abduction problem). Since that, as far as we know, this is not a requirement in GPiD, effectively cvc4sy+b and cvc4sy+u are solving a harder problem than GPiD. We formulated the abduction problem this way, rather than as with all assertions as goals, to avoid trivializing the abduction problem, for which the negation of the goal would always be a solution. Also note that the presence of uninterpreted functions in the abduction problem requires solutions to be generated in a higher-order background logic, which cvc4 supports after a recent extension [7]. As in [19], we used GPiD's version with the Z3 backend. We present their results with (GPiD-1) and without (GPiD) the restriction to limit the set of abduces to size 1.

Table 4. Comparison with GPiD on 400 abduction problems in the QF_UFLIA logic.

	Solved	Unique	Total time
cvc4sy+b	214	0	57290 s
cvc4sy+u	342	0	18735 s
GPiD	193	0	69 s
GPiD-1	398	54	1188 s

Results are summarized in Table 4. cvc4sy+u significantly outperforms cvc4sy+b, both in the number of problems solved and in total time, besides being almost 20% faster on commonly solved problems. We also see that solution finding in GPiD is heavily dependent on which abduces are considered when building solutions, as it solves almost all benchmarks when limited to abduces of size 1 but barely half when unrestricted. It should also be noted that GPiD takes *pre-computed* abduces, whose production time is not accounted for in the evaluation. Despite this, cvc4sy+u is only on average 30% slower on commonly solved problems than GPiD-1 and solves many more problems than GPiD. The big variation of GPiD results in terms of what pre-determined set of candidates can be used in the computation is a severe limitation of their tool. Similarly, while the method proposed in [19] is theory agnostic, their tooling for producing abduces imposes strong limitations on the usage of GPiD for theories other than QF_UFLIA.

[5] At https://github.com/sellamiy/GPiD-Framework.

6 Related Work

The procedure introduced in Sect. 4.2 based on unsat core learning follows a recent trend in enumerative syntax-guided synthesis solving that aims to improve scalability by applying divide-and-conquer techniques, where candidate solutions are built from smaller enumerated pieces rather than being directly enumerated. While previous approaches, both for pointwise [5, 27] and for unrestricted specifications [6], have targeted general-purpose function synthesis, we specialize divide and conquer for solving the abduction problem with a lean (see Sect. 4.4) and effective (see Sect. 5) procedure.

Abductive inference tools for the propositional case include the AbHS and AbHS+ tools [26, 33], based on SAT solvers [11] and hitting set procedures, and the Hyper [23] tool, that includes a series of algorithmic improvements over the former, and uses a MaxSAT solver for computing the hitting set. Like AbHS, GetAbductUCL checks entailment and consistency using two separate calls to the underlying solver, and uses its failures for the selection of new candidates. In contrast, GetAbductUCL keeps this information in two dedicated data structures rather than encoding it with an implicit hitting set. Another significant difference is that the set of hypotheses in the propositional case is fixed and finite, whereas in our setting it is generated dynamically from a grammar. More general approaches, to which our work bears more resemblance and to which we provided an experimental comparison in Sect. 5, are GPiD [19] and Explain [15]. GPiD uses an off-the-shelf SMT solver as a black box to generate ground implicates. It can be used with any theory supported by the underlying SMT solver, similarly to our SyGuS-based approach. While we enumerate formulas that compose the solution for the abduction problem GPiD's authors use *abducibles*, which are equalities and disequalities over the variables in the problem. They similarly build candidates in a refinement loop by combining abducibles according to consistency checks performed by an underlying SMT solver. They use an order on abducibles to guide the search, which is analogous to the enumeration order in enumerative synthesis. Explain on the other hand is built on top of an SMT solver for the theories of linear integer arithmetic and of equality with uninterpreted functions, although its abduction inference procedure in principle can work with any theory that admits quantifier elimination. The method implemented in Explain is based on first determining a subset of the variables in the abduction problem and trying to build the weakest solution over these variables via quantifier elimination, while computing minimal satisfying assignments to ensure that a found solution covers a minimal subset. This method, however, is not complete, as it can miss solutions. The tool also allows the user to specify costs for each variable, so that a given minimal set may be favored.

7 Conclusion

We have described approaches for solving the abduction problem using a modern enumerative solver for syntax-guided synthesis. Our evaluation shows that procedures based on enumerative CEGIS scale for several non-trivial abduction tasks, and have several compelling advantages with respect to other approaches like those used in Explain and GPiD. In several cases, it suffices to use a basic procedure for enumerative CEGIS to generate solutions to abduction problems that are optimal according to certain metrics. Moreover, the generation of feasible solutions can be complemented and accelerated via a procedure for generating conjunctions of enumerated formulas as shown in Fig. 2.

We believe that new abduction capabilities presented in this paper and implemented in cvc4 will be useful in all the applications of abduction we describe in Sect. 3. In addition, we see a number of promising applications in the context of SMT itself. For example, we plan to use abduction to generate useful *conditional rewrite rules* for SMT solvers. Many such rules are used internally by SMT solvers to simplify their input formulas by (equivalence-preserving) term rewriting. The manual identification and selection of good rewrite rules is a tedious and error-prone process. Abduction can be used to generalize a recent approach for the semi-automated development of rewrite rules [28] by synthesizing (most general) conditions under which two terms are equivalent. This in turn can be used to develop new solving strategies in the SMT solver based on those rewrite rules.

References

1. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. ACM SIGPLAN Not. **51**(1), 789–801 (2016)
2. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 1–8. IEEE (2013)
3. Alur, R., Černý, P., Radhakrishna, A.: Synthesis through unification. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 163–179. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_10
4. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: SyGuS-comp 2017: results and analysis. In: Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22 July 2017, pp. 97–115 (2017)
5. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 319–336. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_18
6. Barbosa, H., Reynolds, A., Larraz, D., Tinelli, C.: Extending enumerative function synthesis via SMT-driven classification. In: Barrett, C.W., Yang, J. (eds.) Formal Methods in Computer-Aided Design (FMCAD), pp. 212–220. IEEE (2019)
7. Barbosa, H., Reynolds, A., El Ouraoui, D., Tinelli, C., Barrett, C.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 35–54. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_3
8. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
9. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org

10. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories (chap. 26). In Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
11. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam (2009)
12. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H: Compositional shape analysis by means of bi-abduction. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 289–300 (2009)
13. Cimatti, A., Griggio, A., Sebastiani, R.: Computing small unsatisfiable cores in satisfiability modulo theories. J. Artif. Intell. Res. (JAIR) **40**, 701–728 (2011)
14. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
15. Dillig, I., Dillig, T.: Explain: a tool for performing abductive inference. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 684–689. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_46
16. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. ACM SIGPLAN Not. **47**(6), 181–192 (2012)
17. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. ACM SIGPLAN Not. **48**(10), 443–456 (2013)
18. Dillig, T., Dillig, I., Chaudhuri, S.: Optimal guard synthesis for memory safety. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 491–507. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_32
19. Echenim, M., Peltier, N., Sellami, Y.: A generic framework for implicate generation modulo theories. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 279–294. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_19
20. Echenim, M., Peltier, N., Sellami, Y.: Ilinva: using abduction to generate loop invariants. In: Herzig, A., Popescu, A. (eds.) FroCoS 2019. LNCS (LNAI), vol. 11715, pp. 77–93. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29007-8_5
21. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic Press, Cambridge (2001)
22. Gulwani, S.: Programming by examples: applications, algorithms, and ambiguity resolution. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 9–14. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_2
23. Ignatiev, A., Morgado, A., Marques-Silva, J.: Propositional abduction with implicit hitting sets. In: ECAI 2016–22nd European Conference on Artificial Intelligence, 29 August–2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence, PAIS 2016, pp. 1327–1335 (2016)
24. Josephson, J.R., Josephson, S.G. (eds.): Abductive Inference: Computation, Philosophy, Technology. Cambridge University Press, Cambridge (1994)
25. Li, B., Dillig, I., Dillig, T., McMillan, K., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 370–384. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_26
26. Moreno-Centeno, E., Karp, R.M.: The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. Oper. Res. **61**(2), 453–468 (2013)

27. Neider, D., Saha, S., Madhusudan, P.: Synthesizing piece-wise functions by learning classifiers. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 186–203. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_11
28. Nötzli, A., et al.: Syntax-guided rewrite rule enumeration for SMT solvers. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 279–297. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_20
29. Phothilimthana, P.M., Thakur, A., Bodík, R., Dhurjati, D.: Scaling up superoptimization. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, 2–6 April 2016, pp. 297–310 (2016)
30. Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS language standard version 2.0 (2019)
31. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: cvc4sy: smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 74–83. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_5
32. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_12
33. Saikko, P., Wallner, J.P., Järvisalo, M.: Implicit hitting set algorithms for reasoning beyond NP. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, 25–29 April 2016, pp. 104–113 (2016)
34. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Sarkar, V., Hall, M.W. (eds.) Conference on Programming Language Design and Implementation (PLDI), pp. 281–294. ACM (2005)
35. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Shen, J.P., Martonosi, M. (eds.) Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 404–415. ACM (2006)
36. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013, pp. 287–296 (2013)
37. Zhu, H., Dillig, T., Dillig, I.: Automated inference of library specifications for source-sink property verification. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 290–306. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_21