On Induction for SMT Solvers

Andrew Reynolds and Viktor Kuncak*

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{firstname.lastname}@epfl.ch

Abstract. Satisfiability modulo theory solvers are increasingly being used to solve quantified formulas over structures such as integers and term algebras. Quantifier instantiation combined with ground decision procedure alone is insufficient to prove many formulas of interest in such cases. We present a set of techniques that introduce inductive reasoning into SMT solving algorithms that is sound with respect to the interpretation of structures in SMT-LIB standard. The techniques include inductive strengthening of conjecture to be proven, as well as facility to automatically discover subgoals during an inductive proof, where subgoals themselves can be proven using induction. The techniques have been implemented in CVC4. Our experiments show that the developed techniques have good performance and coverage of a range of inductive reasoning problems. Our experiments also show the impact of different representations of natural numbers and quantifier instantiation techniques on the performance of inductive reasoning. Our solution is freely available in the CVC4 development repository. In addition its overall effectiveness, it has an advantage of accepting SMT-LIB input and being integrated with other SMT solving techniques of CVC4.

1 Introduction

One of the strengths of satisfiability modulo theory (SMT) solvers [3, 8] lies in their efficient handling of many useful theories arising in software verification. These theories often model ubiquitous data types, such as integers, bitvectors, arrays, algebraic data types, sets, or maps. The theories of many of these data types can be naturally thought of as statements that hold in certain concrete structures (for example, integers), or families of structures [17] (for example, lists instantiated into lists of integers). Such semantics is also supported by the SMT-LIB standard's definition of theories [1], and means that the satisfiability of such formulas is determined by its interpretation in these structures, whether or not the satisfiability problem is easily axiomatizable in first-order logic, or whether it is decidable.

From the early days, many SMT solvers and their predecessors have been supporting satisfiability of not only quantifier-free but also universally *quantified* formulas, typically using quantifier instantiation strategies [9], which have become increasingly more robust over time [10, 11, 21]. Quantifiers together with uninterpreted functions mixed with theory symbols give great modeling power to the input language.

* This work is supported in part by the European Research Council (ERC) Project *Implicit Programming*

Unfortunately, the use of quantifier instantiation alone for such problems is highly incomplete, not only in a theoretical sense (the problem is not even recursively enumerable), but also in a very concrete practical sense. Namely, current solvers cannot solve any statements requiring non-trivial use of induction! This is an acknowledged fact in the SMT community. For example, the Z3 tutorial [2] clarifies explicitly that *"The ground decision procedures for recursive datatypes don't lift to establishing inductive facts. Z3 does not contain methods for producing proofs by induction."* Similarly, CVC4 (until now) did not contain a method to perform induction, nor did most other competitive SMT solvers of which we are aware.

Automating induction is a considered very difficult for automated provers [4, 7]. Recent progress has been made in several tools [6,14,15], with which we make detailed comparison in Section 4. Interactive theorem provers heavily use inductive proofs, but have largely avoided to automate induction within their tactics, suggesting that this is among the most difficult tasks to automate. A notable exception is the ALC2 prover, which has early been recognized for its sophisticated inductive reasoning [16]. However, these tools miss an opportunity to fully benefit from efficient theory reasoning: they encode most values using algebraic data types, and need to prove from scratch theory lemmas, which could be handled more efficiently with an SMT approach.

It is worthwhile mentioning that program analysis and verification tools implicitly incorporate inductive reasoning into their algorithms. In fact, it could be argued that the current division of tasks between program analyzers (including software model checkers and verifiers) delegates non-inductive reasoning to SMT solvers, and performs induction in a specialized manner. We do not claim that the techniques we propose will replace such verification techniques, often specialized for the meaning of non-deterministic programs. Instead, we expect that they will complement them, in similar ways that algebraic reasoning of SMT solvers complements fixpoint reasoning of abstract interpretation and software model checking engines. Note that for infinite-state systems, the form of invariants inferred by these tools is often of a particular form, either given by an abstract domain, or given by a class of formulas such as linear constraints [23], or constraint satisfying certain templates [12, 13, 20]. Thefore, especially in cases when invariants themselves may contain recursive functions, it seems desirable to incorporate inductive reasoning into an SMT solver. In fact, Rustan Leino has proposed a pre-processing of formulas to incorporate inductive reasoning, which already proved very helpful for a program verifier based on an SMT solver [19].

In this paper, we present the first technique and implementation of inductive reasoning *within* an SMT solver. Among the advantages of this approach are not only convenience and, in some cases, performance, but also the ability to exploit the internal state of the solver to automatically discover subgoals that themselves need to be proved by induction, which is essential to be able to prove more difficult conjectures.

Contributions. This paper makes the following contributions:

- We describe an approach for supporting inductive reasoning inside an SMT solver that integrates well with existing approaches for handling quantified formulas in SMT. The starting point of this approach is inductively strengthening existentially quantified conjectures.

- We present techniques that help to infer relevant subgoals used in inductive proofs. The generation of subgoals is based on introduction of splitting lemmas into the DPLL(T) framework. The automatically discovered lemmas are generated by enumerating potential equalities while applying the following filtering techniques:
  - limiting the generalization to active terms that refer to variables in the conjecture being proven;
  - inferring universally quantified identities that allow us to remove subgoals that are found to be equivalent to others;
  - removing subgoals that are contradicted by ground facts in the current context.
- We provide a set of 933 benchmarks in the SMT-LIB2 syntax, which are publicly available at `http://lara.epfl.ch/~reynolds/VMCAI2015-ind`. This is the first set of SMT-LIB2 benchmarks targeting inductive reasoning, and includes many of the previously used benchmark sets used to exercise inductive theorem provers.
- We demonstrate that our implementation in the SMT solver CVC4 performs well on this set of benchmarks, in particular through the use of newly developed techniques for inductive reasoning described in this paper. We show our approach is competitive with existing tools for automating induction, comparing favorably against these tools in many cases.

2 Skolemization with Inductive Strengthening

To determine the T-satisfiability of an input set of ground clauses F for some background theory T, a DPLL(T)-based SMT solver first consults a SAT solver for finding a subset of its literals M (which we will call a *context*) that propositionally entails F. If successful, the ground decision procedure for theory T determines the satisfiability of M, adding additional clauses to F as necessary when M is found to be T-unsatisfiable. When extending SMT to quantified formulas, the input F (and likewise a context M) may contain literals whose atoms are universally quantified formulas $\forall x.\ P(x)$.

SMT solvers commonly handle universally quantified formulas $\forall x.P(x)$ from M using instantiation-based techniques, and handle existentially quantified formulas[1] $\neg\forall x.\ P(x)$ from M by *skolemization*. In the latter case, they infer the lemma $(\forall x.\ P(x)) \vee \neg P(k)$, where k is a fresh constant, which is then added to F. We will refer to $\neg P(k)$ as the skolemization of $\neg\forall x.\ P(x)$, and k as the skolem constant for $\neg\forall x.\ P(x)$. Assuming $P(k)$ is quantifier-free, the aforementioned lemma enables a ground decision procedure to reason about the satisfiability of $\neg P(k)$. Unfortunately, SMT solvers have limited ability to prove the unsatisfiability of $\neg P(k)$ in cases when inductive reasoning is required, as in the following example.

Example 1. Assume an axiomatization of the length function $len : List \to Int$:

$$len(nil) \approx 0 \qquad (A_1)$$
$$\forall xy.\ len(cons(x, y)) \approx 1 + len(y) \qquad (A_2)$$

[1] Informally, we refer to $\neg\forall x.\ P(x)$ as an existentially quantified formula, since it is equivalent to $\exists x.\ \neg P(x)$.

and the conjecture $\psi := \forall x.\ len(x) \geq 0$. To determine the satisfiability of $\{A_1, A_2, \neg\psi\}$, the SMT solver by skolemization will add the clause $(\psi \vee \neg len(k) \geq 0)$ to this set for fresh constant k, after which we find a context $\{A_1, A_2, \neg\psi, \neg len(k) \geq 0\}$ that propositionally entails it. The (combined) decision procedure for inductive datatypes and linear arithmetic will determine the satisfiability of the ground porition of this context, namely $\{A_1, \neg len(k) \geq 0\}$, during which it will case split on the constructor for k (either nil or $cons$). In the case where $k \approx nil$, the solver will encounter a conflict noting that $len(nil) \approx 0$. In the case where $k \approx cons(head(k), tail(k))$, the solver may infer the lemma $(\neg A_2 \vee len(k) \approx 1 + len(tail(k)))$ by instantiation. In turn, the solver will find a context containing the ground literals $\{A_1, \neg len(k) \geq 0, len(k) \approx 1 + len(tail(k))\}$, which are satisfied, for instance, by a model where $len(k) \approx -1$ and $len(tail(k)) \approx -2$. Again, the solver may infer by instantiation the lemma $(\neg A_2 \vee len(tail(k)) \approx 1 + len(tail(tail(k))))$, and this loop will continue indefinitely. This is not a coincidence: there exist, in fact, a non-standard model of the axioms used to decide the ground theory of algebraic data types, in which the conjecture is false. In other words, the axioms used within the solver are inadequate for our purpose. □

The aforementioned example can be solved using inductive reasoning. In particular, we may assume without loss of generality that our skolem constant k is the smallest such list that satisfies the property $\neg len(k) \geq 0$, thereby allowing us to assume in particular that $len(tail(k)) \geq 0$. More generally, we may strengthen a conjecture for a variable of sort T when we have a well-founded ordering R over terms of sort T. The general scheme for strengthening our skolemization according to such an R is:

$$(\forall x.\ P(x)) \vee (\neg P(k) \wedge \forall x.\ R(x, k) \Rightarrow P(x)) \tag{1}$$

where k is a fresh constant. We call $\forall x.\ R(x, k) \Rightarrow P(x)$ the *inductive strengthening* of $\neg P(k)$ based on R. Note that conjoining the formula (1) with the initial input formula F does not affect the outcome of the satisfiability of F. The intuition is that if a universal statement does not hold, then there exists the least counterexample with respect to R. Indeed, consider any interpretation for symbols other than k. If $\forall x.\ P(x)$ holds in this interpretation, then the first disjunct of (1) holds in this interpretation. We show that otherwise the second disjunct holds. Consider the set S of all elements y of sort T in this structure such that $\neg P(y)$. Let y_0 any element in S, which exists because $\forall x.\ P(x)$ does not hold. If we consider an arbitrary maximal sequence $y_0, y_1, \ldots \in S$ such that $R(y_{i+1}, y_i)$ for all i, then this sequence must be finite and stop at some y_n, because R is well founded. Let us interpret the fresh constant k as y_n. Then $\neg P(k)$ holds because $y_n \in S$. Because y_n is the last element of the sequence, k also satisfies $\forall x.\ R(x, k) \Rightarrow P(x)$, so the second disjunct of (1) holds. ■

Two examples of well-founded relations R in the context of SMT solving are structural induction for inductive datatypes where $R(s, t)$ if and only if s is a subterm of t, and natural number induction on integers where $R(s, t)$ if and only $0 \leq s < t$. Both of these refer to forms of *strong induction*, where a conjecture is assumed for all terms less than k according to a transitive relation R. Alternatively, we may apply forms of *weak induction*, where for inductive datatypes $R(s, t)$ if and only if s is a direct subterm of t, and for integers $R(s, t)$ if and only if $0 \leq s = t - 1$. The advantage of the weak

form for induction is that, in the case of inductive datatypes, $R(s,t)$ can be encoded without introducing a subterm relation, which is not supported natively by the inductive datatypes theory.

Example 2. The skolemization with inductive strengthening of the negated conjecture $\neg\forall x.\ len(x) \geq 0$ in Example 1 based on weak structural induction is:

$$\neg len(k) \geq 0 \wedge \forall y.(k \approx cons(head(k), tail(k)) \wedge y \approx tail(k)) \Rightarrow len(y) \geq 0)$$

The right conjunct in the formula above simplifies to $k \approx cons(head(k), tail(k)) \Rightarrow len(tail(k)) \geq 0$. With this constraint, the original conjecture can be solved immediately, noting that the length of $tail(k)$ is forced to be non-negative in the case where $k \approx cons(head(k), tail(k))$. □

For quantification over multiple variables, we consider induction schemes that are limited to lexicographic orderings. As a result, we skolemize variables one at a time and independently, starting from the outermost variable. Thus a formula $\neg\forall xy.\ P(x,y)$ is skolemized as: $\forall xy.\ P(x,y) \vee (\neg\forall y.P(k,y) \wedge \forall xy.\ R(x,k) \Rightarrow P(x,y))$. The first conjunct in the conclusion, $\neg\forall y.P(k,y)$, can then be skolemized in the same manner if and when it is necessary to do so. It is also important to note that the variable y is universally quantified in the second conjunct, meaning that $P(x,y)$ can be assumed for any y assuming we choose an x that is smaller k according to R.

For some problems requiring inductive reasoning, it is challenging to determine which variable to apply induction on first. In our approach, the SMT solver is capable of applying induction for different variable orders simultaneously. For instance, in the case of a quantified formula over x and y and induction on y is necessary, this can be done simply by inferring: $\forall xy.P(x,y) \vee \neg\forall yx.P(x,y)$. Subsequently, we will apply induction based on y if and when skolemization is applied to $\neg\forall yx.P(x,y)$.

Our approach is closely related to the approach used in the Dafny tool [19], where (non-negated) conjectures are inductively weakened in an intermediate language before being sent to an SMT solver. Here, we advocate an approach where this transformation is pushed within the core of the SMT solver. This gives several advantages over external approaches. First, the SMT solver may have insight into how and when to invoke inductive strengthening, performing this step lazily or with multiple induction schemes as necessary. Second, certain benchmarks require the skolemization of existentially quantified formulas during the search procedure when a new quantified formula is created or becomes asserted. This may occur, for instance, when instantiating quantified formulas with nested existentially quantified formulas, or in the case when the SMT solver itself introduces an existentially quantified formula of interest, as we will see in the next section. Our approach enables the SMT solver to inductively strengthen its assertions for each such skolemization, which otherwise would not be possible if done externally.

3 Subgoal Generation

A majority of the complexity in inductive reasoning lies in discovering intermediate lemmas, or subgoals, that are required for proving the overall conjecture. A variety of

tools, including [6,14,15], have focused on inferring such subgoals automatically in the context of automated theorem proving. In context of software verification, a subgoal corresponds to a necessary loop invariant or adequate post-condition describing the input/output behavior of a function that is required for a proof to succeed. Tools for this purpose that analyze functional programs include [18,20,22].

In this section, we use the following as a running example.

Example 3. Consider the (combined) theory T of equality and inductively defined datatypes Nat and $List$ whose signature Σ also contains the uninterpreted functions $plus$, app, rev, and sum for natural number addition, list append and reverse, and summing the elements of list. Let $\mathcal{A}$ be the axiomatization of app, rev, and sum where for the latter, $\mathcal{A}$ contains:

$$sum(nil) \approx Z$$
$$\forall xy.\ sum(cons(x,y)) \approx plus(x, sum(y))$$

Now, consider the conjecture $\psi := \forall x.\ sum(rev(x)) \approx sum(x)$. Showing the validity of this conjecture requires, for instance, discovering the intermediate subgoals $\varphi_1 := \forall xy.\ sum(app(x,y)) \approx plus(sum(x), sum(y))$ and $\varphi_2 := \forall xy.\ plus(x,y) \approx plus(y,x)$. Even more so, proving φ_1 itself requires induction and the intermediate subgoal $\varphi_3 := \forall xyz.\ plus(x, plus(y,z)) \approx plus(plus(x,y),z)$.

As we will see in our evaluation, theory reasoning capabilities of the SMT solver can preempt the need for discovering the latter two subgoals φ_2 and φ_3, by enabling the solver to assume that various properties of the builtin integer operator for addition $+$ also hold for applications of the function $plus$. Even so, the SMT solver will not succeed in showing the validity of ψ until it has first discovered and proven φ_1 or some other sufficient subgoal. □

A naive approach for subgoal generation is to enumerate candidate subgoals according to a fair strategy until a set of sufficient subgoals is discovered. In Example 3, we could enumerate all well-typed equalities between Σ-terms built from variables, constructors of sort $List$ and Nat, $plus$, app, rev, and sum up to a particular size until the subgoal φ_1 is discovered. However, an exhaustive enumeration of subgoals is not scalable even for cases where the signature and necessary subgoals are small. It is thus crucial to avoid enumeration of a vast majority of candidate subgoals φ, either by determining that φ is not relevant, redundant, or does not hold.

In this section, we present a design and implementation of an additional component of an SMT solver, which we will refer to as the *subgoal generation module*, whose aim is to discover subgoals that are relevant for proving a given conjecture. We first describe our scheme for basic operation of the subgoal generation module in relation to the rest of the SMT solver, and then describe several heuristics for how it determines which subgoals are likely to be relevant. In particular, these heuristics will make use of the information maintained at the core of a DPLL(T)-based SMT solver. Conceptually, our approach is similar to that of theory enumeration in the Hipspec tool [6], which is based on enumerating candidate subgoals in a principled fashion until a proof for an overall conjecture is found. Like their approach, here we limit ourselves to equality subgoals only. Unlike Hipspec, however, we benefit from integration into a DPLL(T) engine.

```
proc check(F)
  M := findSatAssignment(F)
  if M = fail
    return unsat
  else
    C := getTConflict(M)
    if C = fail
      F := F ∪ quantInst(M) ∪ subgoalGen(M)
  else
      F := F ∪ ¬C
    fi
    return check(F)
```

Fig. 1. The method check, giving the interaction of components within an SMT solver, for an input set of clauses F. The SAT solver (method findSatAssignment), when possible, returns a set of literals M that propositionally entails F. The ground decision procedure(s) (method getTConflict), when possible, returns a subset $C \subseteq M$ that is inconsistent according to the background theory. The quantifier instantiation and subgoal generation modules (methods quantInst and subgoalGen) return a set of clauses based on M.

3.1 Subgoal Generation in DPLL(T)

To prove the conjecture ψ in Example 3, the solver must (1) determine that φ_1 is a relevant subgoal, (2) prove that φ_1 holds, and (3) prove the original conjecture ψ under the assumption φ_1. The DPLL(T) search procedure used by SMT solvers enables a straightforward scheme for accomplishing both (2) and (3). If the subgoal generation module determines that $\forall x.t \approx s$ is a relevant subgoal, it adds $(\neg\forall x.t \approx s) \vee \forall x.t \approx s$, which we refer to as a *splitting lemma*, to the set of clauses currently known by the solver, and additionally may set its decision heuristic to explore the branch $\neg\forall x.t \approx s$ first. A subgoal may be proven by induction, since the skolemization of the assertion $\neg\forall x.\ t \approx s$ can in turn be inductively strengthened according to the method described in Section 2. Subsequently, the solver will backtrack and assert $\forall x.t \approx s$ positively if and only if the standard conflict analysis mechanism of the SMT solver results in $\neg\forall x.\ t \approx s$ to be backtracked during the search. In terms of Example 3, the solver will succeed in proving ψ only after it does so for such a $\forall x.t \approx s$ that entails φ_1. Notice that this behavior is managed entirely by a combination of the SAT solver, ground decision procedures and quantifier instantiation mechanism of the SMT solver, and requires no further intervention from the subgoal generation module, thus enabling it to focus its attention solely on its choice of which subgoals to introduce. This scheme also allows conjecturing multiple candidate subgoals to the system at once, and as needed, during the search, which plays to the advantage of an SMT solver, which is capable of handling inputs having a large number of clauses.

Figure 1 gives the overall interaction between the ground solver, quantifier instantiation, and subgoal generation modules. Notice that the quantifier instantiation and subgoal generation both run after the SAT solver finds a context M which propositionally entails F that is T-consistent according to ground decision procedure(s). Both modules add additional clauses to F in the form of instances of quantified formulas

and splitting lemmas for candidate subgoals respectively. It remains to be shown which subgoals are chosen by the subgoal generation module, i.e. the subgoals in the splitting lemmas returned by the method $\mathsf{subgoalGen}(M)$ for context M.

As mentioned, a naive approach for subgoal generation amounts to a fair enumeration of candidate subgoals. At its core, our approach performs such an enumeration, but discards all candidates that it determines are not useful. For enumerating candidate subgoals in a fair manner, our approach considers subgoals that are smaller than larger ones according to the following measure. Let *size* of a term t be the number of function applications occurring in t plus the number of duplicated variables. For instance, the size of $f(g(x,y))$ is 2, and the the size of $g(x,f(x))$ is 3. The size of a subgoal of the form $\forall \boldsymbol{x}.\ t \approx s$ is the maximum of the size of t and the size of s. Thus, the size of the subgoal φ_1 from Example 3 has size 3. Given a fixed signature Σ, we enumerate the set of all subgoals $\mathcal{S}_n$ of size n, starting with $n = 0$. We will call this the set of *candidate subgoals of size* n. For each n, we heuristically determine a subset $\mathcal{S}_n^R \subseteq \mathcal{S}_n$ of these subgoals, which we will call *relevant* (all others we say are *filtered*). The method $\mathsf{subgoalGen}$ returns splitting lemmas corresponding to a subset of the subgoals $\mathcal{S}_n^R$, where the total number of splitting lemmas it returns does not exceed some fixed number (typically ≤ 3). We continue constructing relevant subgoals for increasing values of n until this limit is reached. In the rest of this section, we will focus on three effective techniques for determining which subgoals are relevant, and which should be filtered.

3.2 Filtering Candidate Subgoals

Filtering based on Active Conjectures Consider the conjecture $\psi := \neg\forall x.\ sum(rev(x)) \approx sum(x)$ from Example 3, and its corresponding skolemization $\neg sum(rev(k)) \approx sum(k)$. An implicit side effect of this skolemization is that a new function symbol k (not occurring in $\mathsf{func}(\Sigma)$) is introduced, thus requiring the solver to determine the satisfiability of constraint in a signature Σ' that extends Σ with k. Assuming all functions in Σ are axiomatized as terminating functions in our axiomatization $\mathcal{A}$, the introduction of k into our constraint is in fact the very reason why inductive reasoning is required, since now the solver cannot reason about Σ'-constraints simply based on a combination of ground theory reasoning and unfolding function definitions by quantifier instantiation. Based on this observation, our first form of filtering is to generate candidate subgoals that state properties about terms that generalize Σ'-terms only, in particular, ones that are not entailed to be equivalent to Σ-terms in the current context.

We say a skolem constant k is *inactive in context* M if $M \models_T k \approx s$ for some Σ-term s, and active otherwise. [2] An existentially quantified formula $\neg\forall x.\ \varphi \in M$ is *inactive in context* M if and only if its skolem constant k is inactive in M, and active otherwise. A term $f(t_1,\ldots,t_n)$ is *inactive in context* M if and only if each of its children is inactive in context M, and active otherwise. For instance, for Example 3, in a context M where $k \approx nil \in M$, we have that k and ψ are inactive, indicating that inductive reasoning is not required for reasoning about the skolemization of ψ in this context. To see this,

[2] Determining whether a skolem constant is active in a context M can be accomplished in the case where k is an inductive datatype, since the decision procedure for inductive datatypes propagates all entailed equalities.

notice that $k \approx nil, \neg sum(rev(k)) \approx sum(k)$ imply $\neg sum(rev(nil)) \approx sum(nil)$, and determining the satisfiability of $\mathcal{A} \wedge \neg sum(rev(nil)) \approx sum(nil)$ can be done by the use of a ground decision procedure and quantifier instantiation for unfolding function definitions.

We say that a Σ-term t is *relevant in context* M if and only if it generalizes an active term s from M, that is, M entails $(t \approx s)\sigma$ for some grounding substitution over $FV(t)$. Notice that since s contains symbols from Σ', all relevant terms are necessarily non-ground. In a context M, the subgoal generation module will only consider conjectures $\forall \boldsymbol{x}.t \approx s$ where t is relevant in M, and $FV(s) \subseteq FV(t)$.

Example 4. Assume a context $M = \{sum(k) \approx Z, sum(rev(k)) \approx S(Z), rev(k) \approx nil\}$. The term $sum(x)$ is relevant in context M since it generalizes the term $sum(k)$, which is active since k is active. The term $sum(rev(x))$ is not relevant in context M since it only generalizes $sum(rev(k))$, but $sum(rev(k))$ is an inactive term in M, since its child $rev(k)$ is inactive. As a result, in context M, the subgoal generation module will filter out all candidate subgoals of the form $\forall x.\ sum(rev(x)) \approx t$. $\square$

To generate the set of all candidate subgoals of size n, we first generate the set $\mathcal{R}_n$ of terms (unique up to variable renaming) of size at most n that are relevant in M, which will be set of terms used on the left-hand side of all candidate subgoals. The set $\mathcal{R}_n$ can be efficient computed by a branching procedure whose states are an (initially empty) sequence of substitutions of the form $(\{x_1 \mapsto t_n\}, \ldots, \{x_n \mapsto t_n\})$ where for each $j = 1, \ldots, n$, either $t_j = x_i$ for some $i \leq j$ or t_j is a well-typed term of the form $f(x_{k+1}, \ldots, x_{k+n})$, where $FV(t_1, \ldots, t_{j-1}) = \{x_1, \ldots, x_k\}$ for $k > j$. Let $term((\sigma_1, \ldots, \sigma_n))$ denote the term $(\ldots(x_1\sigma_1)\ldots)\sigma_n$. Intuitively, appending σ_{n+1} to a state $s = (\sigma_1, \ldots, \sigma_n)$ corresponds to deciding on the form of the subterm x_{n+1} of $term(s)$, either it is a variable or a function applied to new variables not occurring in $term(s)$. We do not explore states s where $term(s)$ has size greater than n, or if $term(s)$ does not generalize an active term from M. Then, $\mathcal{R}_n$ is the set $\{term(s) \mid s \in S\}$ where S is the set of states reached by this procedure.

After several iterations of the loop from Figure 1 on the axiomatization and conjecture from Example 3, we obtain a context M where there are on the order of 20 relevant terms of size 2, and on the order of 100 relevant terms of size 3 that are unique up to variable renaming. Overall in the signature Σ, there are > 40 terms of size at most 2 and > 200 terms of size at most 3 unique up to variable renaming, indicating that this form of filtering determines over half of Σ-terms do not generalize an active term. Notice when Σ contains functions not occurring in the conjecture ψ, the percentage of potential terms this filtering eliminates is even higher.

Filtering based on Canonicity SMT solvers contain efficient methods for reasoning about conjunctions of ground equalities and disequalities, in particular through the use of data structures for maintaining equivalence classes of ground terms, and performing congruence closure over these terms. Note that all inferences (reflexivity, symmetry, transitivity, and congruence) either implicitly or explicitly made by a standard procedure for congruence closure extend to universal equalities as well. Thus, such data structures can be lifted without modification to maintain equivalence classes of non-ground terms that are entailed to be equivalent in a context M.

In detail, say we have a set of equalities $U \subseteq M$ between non-ground terms. In practice, U contains equalities corresponding to (recursive) function definitions from our axiomatization, as well as the set of subgoals we have proven thus far. The subgoal generation module maintains a congruence closure U^* over the set U, where each equivalence class $\{t_1, \ldots, t_n\}$ in U^* is such that M entails $\forall FV(t_i) \cup FV(t_j).t_i \approx t_j$ for each $i, j \in \{1, \ldots, n\}$. The structure U^* can be used to avoid considering multiple conjectures that are effectively equivalent. Each equivalence class in U^* is associated with one of its terms, which we call its *representative term*. We say a term is *canonical in* U^* if and only if it is a representative of an equivalence class in U^*, and *non-canonical in* U^* if and only if it exists in U^* and is not canonical. In our approach, we choose the term in an equivalence class with the smallest size to be its representative term. While enumerating candidate subgoals, we discard all subgoals that contain at least one non-canonical subterm.

Determining whether a subgoal φ is canonical involves adding an equality $t \approx t$ to U for each subterm t of φ not occurring in U^*, and then recomputing U^*. For the purposes of increasing the frequency when a term such as t is found to be non-canonical, we may infer additional equalities between t and terms from U^*, which is based on the following. If we find that $t = s\sigma$ for some substitution σ where s is a term from U^*, and moreover if $s \approx r \in U^*$ and $r\sigma$ is a term from U^*, then we add the equality $t \approx r\sigma$ to U^*, noting that $(s \approx r)\sigma$ is a consequence of $s \approx r$ by instantiation. This allows us to merge the equivalence classes of t and $r\sigma$ in U^*, forcing one of them to be non-canonical, as demonstrated in the following example.

Example 5. Say our context M is $\{\forall x.\ app(x, nil) \approx x\}$. Our universal set of equalities U is $\{app(x, nil) \approx x\}$, and U^* contains the equivalence classes $\{x, app(x, nil)\}$ and $\{nil\}$. Consider a candidate subgoal $\varphi := \forall x.\ rev(app(rev(x), nil))) \approx x$. We add equalities to U for each term in φ that does not yet exist as a term in U^*, after which U^* will additionally contain the equivalence classes $\{rev(x)\}$, $\{app(rev(x), nil)\}$ and $\{rev(app(rev(x), nil))\}$. By unifying these terms with those existing in U^*, we find that $app(rev(x), nil)) = app(x, nil)\sigma$ for substitution $\sigma := \{x \mapsto rev(x)\}$. Since $app(x, nil) \approx x$, and since $x\sigma = rev(x)$, our procedure will merge the equivalence classes $\{rev(x)\}$ and $\{app(rev(x), nil)\}$, to obtain one having $rev(x)$ as its representative term. This indicates that the subgoal $\forall x.\ rev(app(rev(x), nil))) \approx x$ is ***redundant*** in context M, since it contains the non-canonical subterm $app(rev(x), nil)$. We are justified in filtering this subgoal since the above reasoning has determined that it is equivalent to $\forall x.\ rev(rev(x))) \approx x$, which the subgoal generation module may choose to generate instead, if necessary. □

This technique is particularly useful in our approach for subgoal generation in DPLL(T), since our ability to filter candidate subgoals is refined whenever a new subgoal becomes proven. In the previous example, learning the subgoal $\forall x.app(x, nil) \approx x$ allows us to filter an entire class of candidate subgoals, namely that contains a subterm of the form $app(t, nil)$ for any term t. This gives us a constant factor of improvement in our ability to filter future subgoals *for each* subgoal that we prove during the DPLL(T) search.

Filtering based on Ground Facts As mentioned, DPLL(T)-based SMT solvers maintain a context of ground facts M that represent the current satisfying assignment for the

set of clauses F. A straightforward method for determining whether a candidate subgoal $\forall \boldsymbol{x}.\ t \approx s$ does not hold (in M) is to determine if one of its instances is *falsified* by M. In other words, if M entails $\neg(t \approx s)\sigma$, where σ is a grounding substitution over $\boldsymbol{x}$, then clearly $\forall \boldsymbol{x}.\ t \approx s$ does not hold in context M.

Example 6. Assume our context M is $\{\ k \approx nil,\ sum(cons(Z,k)) \approx sum(k),\ sum(k) \approx Z\ \}$, and a candidate subgoal $\varphi := \forall x.\ sum(cons(Z,x)) \approx S(Z)$. We have that M entails $\neg(sum(cons(Z,x)) \approx S(Z))\{x \mapsto nil\}$, indicating that φ does not hold in context M. □

Notice that the fact that φ has a counterexample in context M does not imply that φ will always be filtered, since the SMT solver may later find a different context that does not contain $sum(k) \approx Z$. Conversely, we may choose to filter candidate subgoals $\forall \boldsymbol{x}.\ t \approx s$ if none (or fewer than some constant number) of its instances are entailed in the context M, that is, M does not entail $(t \approx s)\sigma$ for any grounding substitution over $\boldsymbol{x}$. Note the following example.

Example 7. Assume our context M is $\{\ sum(cons(Z,k)) \approx plus(Z, sum(k)),\ plus(Z, sum(k)) \approx sum(k)\ \}$, and a candidate subgoal $\varphi := \forall x.\ sum(x) \approx S(Z)$. Although no ground instance of φ is falsified, neither is any ground instance of φ entailed. Thus, we may choose to filter φ. □

To give a rough and informal idea of the overall number of subgoals that are filtered by these techniques, consider the axiomatization and conjecture ψ from Example 3. We found that there were approximately 6230 well-typed equalities between Σ-terms that met the basic syntactic requirements of being a candidate subgoal[3]. We measured the average number of relevant subgoals for contexts M obtained after several iterations of the loop from Figure 1. With filtering based on active conjectures alone, there were on average approximately 4180 relevant subgoals of size at most 3, with filtering based on canonicity alone (given only the initial set of axioms in $\mathcal{A}$), there were approximately 4900, and with filtering based on ground facts alone, there were approximately 1200. With all three filtering techniques enabled, there were approximately 800 relevant subgoals of size at most 3, reducing the space of conjectures over seven times. Furthermore, filtering based on the canonicity of the candidate subgoal is refined whenever a new subgoal becomes proven. We thus found that, once the solver proves the commutativity and right identity of $plus$, as well as the right identity of app, the number of relevant subgoals of size at most 3 decreased to around 350 on average, making the discovery of the sufficient subgoal φ_1 in this example much less daunting from a practical perspective.

4 Evaluation

We have implemented the techniques described in this paper in the SMT solver CVC4 [3]. We evaluate the implementation on a library of 933 benchmarks, which we

[3] Namely, for a subgoal $\forall \boldsymbol{x}.\ t \approx s$, we require $FV(s) \subseteq FV(t)$, and t must be an application of an uninterpreted function.

constructed from several sources, including previous test suites for tools that specifically target induction (Isaplanner, Clam, Hipspec), as well as verification conditions from the Leon verification system. The benchmarks in SMT-LIB2 format can be retrieved from `http://lara.epfl.ch/~reynolds/VMCAI2015-ind`.

Isaplanner. We considered 85 benchmarks from the test suite for automatic induction introduced by the authors of the Isaplanner system [15]. These benchmarks contain conjectures involving lists, natural numbers, and binary trees. A handful of these benchmarks involved higher-order functions on lists, such as map, which we encoded using an auxiliary uninterpreted function as input (the function to be mapped) for each instance of map in a conjecture.

Clam. We considered 86 benchmarks used for evaluating the CLAM prover [14]. Of the 86 benchmarks, 50 are conjectures designed such that subgoal generation is likely necessary for the proof to succeed, 12 are generalizations of these conjectures, and 24 are subgoals that were discovered by CLAM during its evaluation. These benchmarks involve lists, natural numbers, and sets.

Hipspec. We considered benchmarks based on three examples from [6], which included intermediate subgoals used by the HipSpec theorem prover for proving various conjectures. The first example states that list reverse is equivalent to its tail-recursive version, the second example states that rotating a list by its length returns the original list, and the third example states that the sum of the first n cubes is the n^{th} triangle number squared. Between the three examples, there are a total of 26 benchmarks, 16 of which are reported to require subgoals.

Leon. We considered three sets of benchmarks for programs taken from Leon, a system for verification and synthesis of Scala programs (`http://lara.epfl.ch/w/leon`). We considered these benchmarks since they involve more sophisticated data structures (such as queues, binary trees and heaps), and are representative of properties seen when verifying simple functional programs. In the first set, we conjecture the correctness of various operations on amortized queues, in particular that enqueue and pop behave analogously to a corresponding implementation on lists. In the second set (see the Appendix), we conjecture the correctness of various operations on binary search trees, in particular that membership lookup according to binary search is correct if the tree is sorted, and the correctness of removing an element from a tree.

4.1 Encodings

For our evaluation, we considered three encodings of the aforementioned benchmarks into SMT-LIB2 syntax. In the first encoding, which we will refer to as **dt**, all functions were encoded as uninterpreted functions over inductive datatypes. In particular, natural numbers were encoded as an inductive datatype with constructors S and Z, and sets were represented using the same datatype for lists, where its constructors $cons$ and nil represented insertion and the empty set respectively.

Direct Translation to Theory. For the purposes of leveraging the decision procedures of the SMT solver for reasoning about the behavior of built-in functions, we considered an alternative encoding, which we will refer to as **dtt**. This encoding is obtained as a

result of replacing all occurrences of certain datatypes with builtin sorts. For instance, we replace all occurrences of Nat (the datatype for natural numbers) with Int (the built-in type for integers) according to the following steps. First, all occurrences of f-applications are replaced by f_i-applications where f_i is an uninterpreted function whose sort is obtained from the sort of f by replacing all occurrences of Nat by Int. All variables of sort Nat in quantified formulas are replaced by variables of sort Int. All occurrences of $S(t)$ are replaced by $1 + t$ (where $+$ is the built-in operator for integer addition), and all occurrences of Z were replaced by the integer numeral 0. Second, to preserve the semantics of natural numbers, all quantified formulas of the form $\forall x.\varphi$ where x is of type Int are replaced with $\forall x.x \geq 0 \Rightarrow \varphi$ (indicating a pre-condition for the function/conjecture), and for all functions $f_i : S_1 \times \ldots \times S_n \rightarrow Int$, the quantified formula $\forall x_1, \ldots, x_n.f_i(x_1, \ldots, x_n) \geq 0$ was added (indicating a post-condition for the function). Finally, constraints are added, wherever possible, stating the equivalence between uninterpreted functions from Σ and a corresponding built-in functions supported by the SMT solver if one existed. For instance, we add the quantified formulas $\forall xy.\ (x \geq 0 \wedge y \geq 0) \Rightarrow plus(x, y) = x + y$ and $\forall xy.\ (x \geq 0 \wedge y \geq 0) \Rightarrow less(x, y) \Leftrightarrow x < y$. [4] Since CVC4 has recently added support for a native theory for sets, a similar translation was done for set operations as well, so insertion and empty data structure are replaced by $\{x\} \cup y$ and $\emptyset$, respectively.

Datatype to Theory Isomorphism. We considered a third encoding, which we will refer to **dti**, that is intended to capitalize on the advantages of both encodings **dt** and **dtt**. In this encoding, we use the signature Σ, axioms for function definitions, and all conjectures as for **dtt**, and introduce uninterpreted functions to map between certain datatypes and builtin types. For instance, we introduce an uninterpreted function $f_{Nat} : Nat \rightarrow Int$ mapping natural numbers as algebraic data type into the built-in integer type. We add constraints to all benchmarks for its definition, also stating that f_{Nat} is an injection to non-negative integers:

$$f_{Nat}(Z) \approx 0 \qquad \forall x.\ f_{Nat}(S(x)) \approx 1 + f_{Nat}(x)$$
$$\forall x.\ f_{Nat}(x) \geq 0 \quad \forall xy.\ f_{Nat}(x) \approx f_{Nat}(y) \Rightarrow x \approx y$$

We then add constraints for the uninterpreted functions from Σ that correspond to built-in functions involving Int that are supported by the solver. For instance, we add the constraints $\forall xy.\ f_{Nat}(plus(x, y)) \approx f_{Nat}(x) + f_{Nat}(y)$ and $\forall xy.\ f_{Nat}(less(x, y)) \Leftrightarrow f_{Nat}(x) < f_{Nat}(y)$. A similar mapping was introduced between lists and sets, where constraints were added for each basic set operation.

4.2 Results

In our results, we evaluate the performance of our implementation in the SMT solver CVC4 on all benchmarks in each of the three encodings. To measure the number of benchmarks that can be solved without inductive reasoning, we ran the SMT solver Z3 [8], as well as CVC4 without the inductive reasoning module enabled (as indicated

[4] We did not provide this constraint for multiplication $mult$, since it introduces non-linear arithmetic, which SMT solvers only have limited support for.

Encoding	Config	**Isaplanner** 85	**Clam+sg** 86	**Clam** 50	**Hipspec+sg** 26	**Hipspec** 16	**Leon+sg** 46	**Total** 311
dt	**z3**	16	11	0	2	0	6	35
	cvc4	15	5	0	4	0	7	31
	cvc4+i	69	73	7	26	3	29	207
	cvc4+ig	73	75	25	25	5	34	237
dtt	**z3**	35	19	4	4	1	9	72
	cvc4	34	15	2	5	1	8	65
	cvc4+i	64	61	5	16	3	37	186
	cvc4+ig	65	62	9	16	3	37	192
dti	**z3**	35	22	3	5	1	9	75
	cvc4	34	17	3	6	1	9	70
	cvc4+i	77	79	14	26	6	41	243
	cvc4+ig	79	80	28	25	8	40	260

Fig. 2. Number of solved benchmarks. All experiments run with a 300 second timeout. The suffix $+sg$ indicates classes where subgoals were explicitly provided. All benchmarks in the **Clam** and **Hipspec** classes are reported to require subgoals. The **Isaplanner** class contains a mixture of benchmarks, some of which require subgoals.

by the configuration **cvc4**). [5] We then ran two configurations of CVC4 with inductive reasoning. The first, configuration **cvc4+i** is identical to the behavior of CVC4, except that it applies skolemization with inductive strengthening as described in Section 2. The second configuration **cvc4+ig** additionally enables the subgoal generation scheme as described in Section 3. In both configurations, inductive strengthening is applied to all inductive datatype skolem variables based on weak structural induction, and to all integer skolem variables based on weak natural number induction. All configurations of CVC4 used newly developed quantifier instantiation techniques that prioritize instantiations that lead to ground conflicts [21].

Figure 2 shows the results for the four configurations on each of the three encodings. For isolating the benchmarks where subgoal generation is reported to be necessary, we divide the results for the Clam and Hipspec classes into two columns. The first (columns **Clam+sg** and **Hipspec+sg**) explicitly provide all necessary subgoals (if any) as indicated by the sources of the benchmarks in [14] and [6] as theorems. The second (columns **Clam** and **Hipspec**) includes only the benchmarks where subgoals were required, and does not explicitly provide these subgoals. The Leon benchmarks were considered sequentially: to prove k^{th} conjecture, the previous $k-1$ conjectures were assumed as theorems for the next conjecture, whether they were needed or not. Therefore, these benchmarks contain many quantified assumptions.

As expected, a majority of the benchmarks over all classes in the base encoding **dt** require inductive reasoning, as Z3 and CVC4 solve 35 and 31 respectively (around 10% of the benchmarks overall). Encodings that incorporate theory reasoning eliminated the need for inductive reasoning for approximately an additional 10% of the benchmarks, as Z3 and CVC4 solve 73 and 66 respectively on benchmarks in the **dtt** encoding, and 76 and 71 respectively in the **dti** encoding.

Our results show that the basic configuration of inductive reasoning **cvc4+i** has a relatively high success rate for classes where subgoal generation is reported to be unnec-

[5] Note these two configurations were only run to measure the number of benchmarks that did not require inductive reasoning, and not to be considered as competitive.

essary (**Clam+sg**, **Hipspec+sg** and **Leon+sg**). Over these three sets, **cvc4+i** solves 128 (81%) of the benchmarks in the **dt** encoding, 114 (72%) in the **dtt** encoding, and 146 (92%) in the **dti** encoding. We found that 5 of the heapsort benchmarks from **Leon+sg** required an induction scheme based on induction on the size of a heap, consequently **cvc4+i** (as well as **cvc4+ig**) was unable to solve them. Our results confirm that subgoal generation is necessary for a majority of benchmarks in the **Clam** and **Hipspec** classes, as **cvc4+i** solves only 10 out of 66 total in these sets. [6] However, note that **cvc4+i** solves twice as many of these benchmarks (20) simply by leveraging theory reasoning, as seen in the results for **Clam** and **Hipspec** in the **dti** encoding.

With subgoal generation enabled, CVC4 was able to solve an additional 53 benchmarks over all classes and encodings. In total, CVC4 automatically inferred subgoals sufficient for proving conjectures in 57 cases that were otherwise unsolvable without subgoal generation. This improvement was most noticeable on the benchmarks from the **dt** encoding, where **cvc4+ig** solved 30 more than **cvc4+i** (237 vs. 207). This can be attributed to the fact that many of the subgoals it discovered related to simple facts related to arithmetic functions, such as the commutatitivity and associativity of $plus$, whereas in the other two encodings these facts are inherent consequences of theory reasoning. The performance of the subgoal generation module was the least noticable on benchmarks from the **dtt** encoding, which we attribute to the fact that the techniques from Section 3 are not well suited for signatures that contain theory symbols. In the **dti** encoding, subgoal generation led to **cvc4+ig** solving 17 more benchmarks than **cvc4+i** (260 vs. 243). The techniques for filtering candidate subgoals from Section 3.2 were critical for these cases. We found that only 2 of these 17 benchmarks were solved in a configuration identical to **cvc4+ig** but where all filtering techniques were disabled.

We remark that **cvc4+ig** was able to discover and prove several interesting subgoals for these benchmarks. For the conjecture $\forall nx.\ count(n, x) \approx count(n, sort(x))$ from the **Isaplanner** class, stating that the number of times n occurs in a list is the same after an insertion sort, we first determined by paper-and-pencil analysis that this would need two subgoals (also occurring in the Isaplanner set):

$$\forall nx.\ count(n, insert(n, x)) \approx S(count(n, x)),\ \text{and}$$
$$\forall nmx.\ \neg n \approx m \Rightarrow count(n, insert(m, x)) \approx count(n, x)$$

However, CVC4's subgoal generation module found and proved a single subgoal $\forall nmx.\ count(n, insert(m, x)) \approx count(n, cons(m, x))$, which by itself was sufficient to prove the original conjecture. CVC4 was thus able to fully automatically find a simpler proof than we did by hand.

On a majority on the benchmarks we considered, the subgoal generation module has only a small overhead in performance for benchmarks where subgoal generation is not required. In only 17 cases **cvc4+ig** took more than twice as long to solve a benchmark

[6] These 10 benchmarks are solved by CVC4 without subgoal generation, despite being described in literature as requiring subgoals. In some cases, the reason is that CVC4 chose a different variable to apply induction to. For instance, the conjecture $rotate(S(n), rotate(m, xs)) \approx rotate(S(m), rotate(n, xs))$ is said to be proven by Hipspec by induction on xs after discovering the subgoal $rotate(n, rotate(m, xs)) \approx rotate(m, rotate(n, xs))$. Instead, CVC4 proved this conjecture by induction on n using no subgoals.

Id	Property	Solved only by
47	$\forall t.\ height(mirror(t)) = height(t)$	CVC4, HipSpec, Zeno
50	$\forall x.\ butlast(x) = take(minus(len(x), S(Z)), x)$	CVC4, Zeno
54	$\forall mn.\ minus(plus(m, n), n) = m$	CVC4, HipSpec, Zeno
56	$\forall nmx.\ drop(n, drop(m, x)) = drop(plus(n, m), x)$	CVC4, HipSpec, Zeno
66	$\forall x.\ leq(len(filter(x)), len(x))$	CVC4, ACL2, Zeno
67	$\forall x.\ len(butlast(x)) = minus(len(x), S(Z))$	CVC4, HipSpec, Zeno
68	$\forall xl.\ leq(len(delete(x, l)), len(l))$	CVC4, ACL2, Zeno
81	$\forall nmx.\ take(n, drop(m, x)) = drop(m, take(plus(n, m), x))$	CVC4, HipSpec, Zeno
83	$\forall xyz.\ zip(app(x, y), z) = app(zip(x, take(len(x), z)), zip(y, drop(len(x), z)))$	CVC4, HipSpec, Zeno
84	$\forall xyz.\ zip(x, app(y, z)) = app(zip(take(len(y), x)y), zip(drop(len(y), x), z))$	CVC4, HipSpec, Zeno
52	$\forall nl.\ count(n, l) = count(n, rev(l))$	ACL2, HipSpec, Zeno
72	$\forall ix.\ rev(drop(i, x)) = take(minus(len(x), i)rev(x))$	Hipspec
73	$\forall x.\ rev(filter(x)) = filter(rev(x))$	HipSpec, Zeno
74	$\forall ix.\ rev(take(i, x)) = drop(minus(len(x), i)rev(x))$	Hipspec
78	$\forall l.\ sorted(sort(l))$	ACL2, Zeno
85	$\forall xy.\ len(x) = len(y) \Rightarrow zip(rev(x), rev(y)) = rev(zip(x, y))$	

Fig. 3. Isaplanner benchmarks that cannot be solved by either a competing inductive prover, or using CVC4 with its inductive mode with subgoal generation on the **dti** encoding. The first part shows benchmarks solved by our approach but not by one of the competing provers. Zeno excels at these benchmarks, but note that, e.g., CVC4 solves 17 Clam benchmarks that Zeno cannot.

than **cvc4+i** (for benchmarks that took **cvc4+ig** more than a second to solve), and in only 4 cases **cvc4+ig** was unable to solve a benchmark that **cvc4+i** solved.

Overall, the results show that the performance of all configurations is the best for benchmarks in the **dti** encoding. While the **dtt** encoding enables the SMT solver to leverage the decision procedure for linear integer arithmetic when reasoning about inductive conjectures, it degrades performance for many benchmarks, often leading to conjectures being unsolved. We attribute this to several factors. Firstly, the **dtt** encoding complicates the operation of the matching-based heuristic for quantifier instantiation. For instance, finding ground terms that modulo equality match a pattern $f(1+x)$ is less straightforward than finding terms that match a pattern $f(S(x))$. Secondly, as opposed to the other two encodings, the **dtt** encoding relies heavily on decisions made by the theory solver for linear integer arithmetic. For a negated conjecture $\neg P_i(k_i)$ for integer k_i, a highly optimized Simplex decision procedure for linear integer arithmetic will find a satisfying assignment, which may or may not choose to explore useful values of k_i. On the other hand, given a negated conjecture $\neg P(k)$ for natural number k, in the absense of conflicts, the decision procedure for inductive datatypes will first case-split on whether k is zero. We believe the behavior of the decision procedure for inductive datatypes has more synergy with the quantifier instantiation mechanism in CVC4 for our axiom sets, since its case splitting naturally corresponds with the case splitting in the definition of recursive functional programs. As a result, the **dti** encoding is the best of the three, as it allows the solver to effectively consult the integer solver for making theory-specific inferences as needed, without affecting the interaction between the ground solver and quantifier instantiation mechanism.

Comparison with Inductive Theorem Provers. By comparing to reported results of inductive provers on different benchmarks, we find that tools perform well on their own benchmark sets, but, unsurprisingly, less well on benchmarks used to evaluate competing tools. Although no tool dominates, **cvc4+ig** performs reasonably well across

different benchmark sets. Combined with the convenience of using the standardized SMT-LIB2 format and the benefits of other SMT techniques, CVC4 becomes an attractive choice for inductive proofs.

For the 85 benchmarks in Isaplanner set, **cvc4+ig** solves a total of 79 benchmarks in the **dti** encoding. These benchmarks have been translated into the native formats supported by a number of tools. As points of comparison, as reported in [24], Zeno solves a total of 82 benchmarks, 3 that **cvc4+ig** cannot. Hipspec [6] solves a total of 80 benchmarks, 4 that **cvc4+ig** cannot, while **cvc4+ig** solves 3 benchmarks that Hipspec cannot. ACL2 [5] solves a total of 73 benchmarks, 2 that **cvc4+ig** cannot, while **cvc4+ig** solves 8 that ACL2 cannot. We list all benchmarks that either CVC4, Zeno, Hipspec, or ACL2 does not solve in Figure 3. Isaplanner [15] and Dafny [19] do not incorporate techniques for automatically generating subgoals, and solve 47 and 45 benchmarks respectively. Interestingly, we found that one property in the original set of benchmarks from [15], $\forall xyz.\, less(x,y) \Rightarrow mem(x, insert(y,z)) \approx mem(x,z)$ is true, although it is cited in later sources as not a theorem, and excluded from the evaluation of the other tools. We found that CVC4 was able to prove this property, both when theory reasoning was incorporated (**cvc4+i** and **cvc4+ig** on the **dtt** and **dti** encodings), as well as when subgoal generation was enabled (**cvc4+ig** on the **dt** encoding).

For the original 50 benchmarks from the Clam set (which include 38 benchmarks from **Clam** class in Figure 2 that require subgoals and 12 benchmarks from the **Clam+sg** class that do not), **cvc4+ig** solves a total of 34 benchmarks in the **dti** encoding. A version of Hipspec solves a total of 47 of these benchmarks, 15 that **cvc4+ig** cannot, while **cvc4+ig** solves 2 benchmarks that Hipspec cannot (these 2 benchmarks were solved due to the use of CVC4's native support for sets). Zeno solves a total of 21 benchmarks, 4 that **cvc4+ig** cannot, while **cvc4+ig** solves 17 that Zeno cannot. The Clam tool itself solves 36 fully automatically, 7 that **cvc4+ig** cannot, while **cvc4+ig** solves 1 that Clam cannot, namely proving that list reverse is equivalent to its tail recursive version.

5 Conclusion

We have presented a method for incorporating inductive reasoning within a DPLL(T)-based SMT solver. We have shown an implementation that has a high success rate for benchmarks taken from automated theorem proving and software verification sources, and is competitive with state-of-the-art tools for automating induction. We have provided a larger and unified set of benchmarks in a standard SMT-LIB2 format, which will make future comparisons and competitions feasible, including the analysis of running times of tools. Our evaluation indicates the inductive reasoning capabilities in our approach benefit from an encoding where theory reasoning can be consulted using a mapping between datatypes and builtin types, allowing the SMT solver to leverage inferences made by its ground decision procedures. Our evaluation shows that our approach for subgoal generation is feasible for automatically inferring subgoals that are relevant to proving a conjecture. The scalability of our approach is made possible by several powerful techniques for filtering irrelevant candidate subgoals based on the information the solver knows about its current context. Future work includes incorporat-

ing further induction schemes, inferring subgoals containing propositional symbols, and improvements to the heuristics used for enumerating and filtering candidate subgoals.

References

1. SMT-LIB theories, 2014. `http://smtlib.cs.uiowa.edu/theories.shtml`.
2. Z3 will not prove inductive facts, September 2014. `http://rise4fun.com/z3/tutorial`.
3. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification (CAV)*, pages 171–177, 2011.
4. A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning (Volume 1)*, chapter 13. Elsevier and The MIT Press, 2001.
5. H. R. Chamarthi, P. C. Dillinger, P. Manolios, and D. Vroon. The ACL2 Sedan theorem proving system. In *TACAS*, 2011.
6. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *CADE*, 2013.
7. H. Comon. Inductionless induction. In *Handbook of Automated Reasoning (Volume 1)*, chapter 14. Elsevier and The MIT Press, 2001.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
9. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
10. C. Flanagan, R. Joshi, and J. B. Saxe. An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Laboratories Palo Alto, 2004.
11. Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE*, 2007.
12. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
13. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *Programming Languages and Systems - 9th Asian Symposium, APLAS*, 2011.
14. A. Ireland. Productive use of failure in inductive proof. *J. Autom. Reasoning*, 16(1-2):79–111, 1996.
15. M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *Interactive Theorem Proving (ITP)*, 2010.
16. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
17. S. Krstic, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *TACAS*, volume 4424 of *LNCS*, pages 602–617, 2007.
18. R. Ledesma-Garza and A. Rybalchenko. Binary reachability analysis of higher order functional programs. In *Static Analysis Symposium (SAS)*, 2012.
19. K. R. M. Leino. Automating induction with an SMT solver. In *VMCAI*, 2012.
20. R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In *Computer Aided Verification (CAV)*, 2014.
21. A. Reynolds, C. Tinelli, and L. D. Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2014.
22. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.
23. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *Computer Aided Verification (CAV)*, 2013.
24. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, 2012.

APPENDICES

A Axioms Defining Binary Search Tree Operations

The following section presents, using SMT-LIB2 format, the definitions of binary search tree operations, expressed as universally quantified axioms constraining otherwise uninterpreted function symbols. Symbols $\neg, \forall, \Rightarrow$ stand for not, forall, =>.

```
; natural numbers
(declare-datatypes () ((Nat (succ (pred Nat)) (zero))))

(declare-fun less (Nat Nat) Bool)
(assert (¬ (less zero zero)))
(assert (∀ ((x Nat)) (less zero (succ x))))
(assert (∀ ((x Nat) (y Nat)) (= (less (succ x) (succ y)) (less x y))))
(define-fun leq ((x Nat) (y Nat)) Bool (or (= x y) (less x y)))

(declare-fun plus (Nat Nat) Nat)
(assert (∀ ((n Nat)) (= (plus zero n) n)))
(assert (∀ ((n Nat) (m Nat)) (= (plus (succ n) m) (succ (plus n m)))))

(declare-fun nmax (Nat Nat) Nat)
(assert (∀ ((n Nat) (m Nat)) (= (nmax n m) (ite (less n m) m n))))

(declare-datatypes () ((Lst (cons (head Nat) (tail Lst)) (nil)))); Lists

(declare-fun append (Lst Lst) Lst)
(assert (∀ ((x Lst)) (= (append nil x) x)))
(assert (∀ ((x Nat) (y Lst) (z Lst)) (= (append (cons x y) z) (cons x (append y z)))))

(declare-fun len (Lst) Nat)
(assert (= (len nil) zero))
(assert (∀ ((x Nat) (y Lst)) (= (len (cons x y)) (succ (len y)))))

(declare-fun mem (Nat Lst) Bool)
(assert (∀ ((x Nat)) (¬ (mem x nil))))
(assert (∀ ((x Nat) (y Nat) (z Lst)) (= (mem x (cons y z)) (or (= x y) (mem x z)))))

; binary search tree
(declare-datatypes () ((Tree (node (data Nat) (left Tree) (right Tree)) (leaf))))

(declare-fun tinsert (Tree Nat) Tree)
(assert (∀ ((i Nat)) (= (tinsert leaf i) (node i leaf leaf))))
(assert (∀ ((r Tree) (l Tree) (d Nat) (i Nat))
   (= (tinsert (node d l r) i) (ite (less d i) (node d l (tinsert r i)) (node d (tinsert l i) r)))))

(declare-fun height (Tree) Nat)
(assert (= (height leaf) zero))
(assert (∀ ((x Nat) (y Tree) (z Tree))
```

```
        (= (height (node x y z)) (succ (nmax (height y) (height z))))))

(declare-fun tinsert-all (Tree Lst) Tree)
(assert (∀ ((x Tree)) (= (tinsert-all x nil) x)))
(assert (∀ ((x Tree) (n Nat) (l Lst))
        (= (tinsert-all x (cons n l)) (tinsert (tinsert-all x l) n))))

(declare-fun tsize (Tree) Nat)
(assert (= (tsize leaf) zero))
(assert (∀ ((x Nat) (l Tree) (r Tree))
        (= (tsize (node x l r)) (succ (plus (tsize l) (tsize r))))))

(declare-fun tremove (Tree Nat) Tree)
(assert (∀ ((i Nat)) (= (tremove leaf i) leaf)))
(assert (∀ ((i Nat) (d Nat) (l Tree) (r Tree)) (⇒ (less i d)
        (= (tremove (node d l r) i) (node d (tremove l i) r)))))
(assert (∀ ((i Nat) (d Nat) (l Tree) (r Tree)) (⇒ (less d i)
        (= (tremove (node d l r) i) (node d l (tremove r i))))))
(assert (∀ ((d Nat) (r Tree)) (= (tremove (node d leaf r) d) r)))
(assert (∀ ((d Nat) (ld Nat) (ll Tree) (lr Tree) (r Tree))
        (= (tremove (node d (node ld ll lr) r) d) (node ld (tremove (node ld ll lr) ld) r))))

(declare-fun tremove-all (Tree Lst) Tree)
(assert (∀ ((x Tree)) (= (tremove-all x nil) x)))
(assert (∀ ((x Tree) (n Nat) (l Lst))
        (= (tremove-all x (cons n l)) (tremove-all (tremove x n) l))))

(declare-fun tcontains (Tree Nat) Bool)
(assert (∀ ((i Nat)) (¬ (tcontains leaf i))))
(assert (∀ ((d Nat) (l Tree) (r Tree) (i Nat))
        (= (tcontains (node d l r) i) (or (= d i) (tcontains l i) (tcontains r i)))))

(declare-fun tsorted (Tree) Bool)
(assert (tsorted leaf))
(assert (∀ ((d Nat) (l Tree) (r Tree)) (= (tsorted (node d l r))
            (and (tsorted l) (tsorted r)
                (∀ ((x Nat)) (⇒ (tcontains l x) (leq x d)))
                (∀ ((x Nat)) (⇒ (tcontains r x) (less d x))))))))

(declare-fun tmember (Tree Nat) Bool)
(assert (∀ ((x Nat)) (¬ (tmember leaf x))))
(assert (∀ ((d Nat) (l Tree) (r Tree) (i Nat))
        (= (tmember (node d l r) i) (ite (= i d) true (tmember (ite (less d i) r l) i)))))

(declare-fun content (Tree) Lst)
(assert (= (content leaf) nil))
(assert (∀ ((d Nat) (l Tree) (r Tree))
        (= (content (node d l r)) (append (content l) (cons d (content r))))))
```

B Tree Properties Proved Automatically

This section lists some of the conjectures about the operations defined in the previous section, proved fully automatically by our CVC4 extension.

(∀ ((t Tree) (n Nat)) (= (tsize (tinsert t n)) (succ (tsize t))))
(∀ ((l Lst) (t Tree)) (leq (tsize t) (tsize (tinsert−all t l))))
(∀ ((l Lst) (t Tree)) (= (tsize (tinsert−all t l)) (plus (tsize t) (len l))))
(∀ ((t Tree) (n Nat)) (leq (tsize (tremove t n)) (tsize t)))
(∀ ((l Lst) (t Tree)) (leq (tsize (tremove−all t l)) (tsize t)))
(∀ ((x Tree) (i Nat)) (tcontains (tinsert x i) i)))
(∀ ((i Nat) (x Tree) (j Nat)) (= (**or** (= i j) (tcontains x j)) (tcontains (tinsert x i) j)))
(∀ ((x Tree) (i Nat)) (⇒ (tsorted x) (tsorted (tinsert x i))))
(∀ ((x Tree) (i Nat)) (tmember (tinsert x i) i))
(∀ ((i Nat) (x Tree) (j Nat)) (= (**or** (= i j) (tmember x j)) (tmember (tinsert x i) j)))
(∀ ((i Nat) (x Tree)) (⇒ (tsorted x) (= (tcontains x i) (tmember x i))))
(∀ ((i Nat) (x Tree)) (⇒ (tmember x i) (tcontains x i)))
(∀ ((l Lst) (x Tree) (n Nat)) (= (tinsert−all (tinsert x n) l)
(tinsert−all x (append l (cons n nil)))))
(∀ ((x Lst)) (tsorted (tinsert−all leaf x)))
(∀ ((x Lst) (i Nat)) (= (mem i x) (tcontains (tinsert−all leaf x) i)))
(∀ ((x Lst) (y Lst) (i Nat)) (= (mem i (append x y)) (**or** (mem i x) (mem i y))))
(∀ ((x Tree) (i Nat)) (⇒ (tsorted x) (= (tmember x i) (mem i (content x)))))
(∀ ((x Tree) (i Nat)) (= (tcontains x i) (mem i (content x))))