Can LLM Aid in Solving Constraints with Inductive Definitions?

Weizhi Feng[1,2][0000−0003−0710−223X], Shidong Shen[1,2][0009−0000−0369−021X], Jiaxiang Liu[1,2][0000−0002−6725−8167], Taolue Chen[3][0000−0002−5993−1665], Fu Song[1,2,4][0000−0002−0581−2679], and Zhilin Wu[1,2(✉)][0000−0003−0899−628X]

[1] Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Birkbeck, University of London, United Kingdom
[4] Nanjing Institute of Software Technology, Nanjing, China
`{fengwz,shensd,liujx,songfu,wuzl}@ios.ac.cn` `t.chen@bbk.ac.uk`

Abstract. Solving constraints involving inductive (aka recursive) definitions is challenging. State-of-the-art SMT/CHC solvers and first-order logic provers provide only limited support for solving such constraints, especially when they involve, e.g., abstract data types. In this work, we leverage structured prompts to elicit Large Language Models (LLMs) to generate auxiliary lemmas that are necessary for reasoning about these inductive definitions. We further propose a neuro-symbolic approach, which synergistically integrates LLMs with constraint solvers: the LLM iteratively generates conjectures, while the solver checks their validity and usefulness for proving the goal. We evaluate our approach on a diverse benchmark suite comprising constraints originating from algebraic data types and recurrence relations. The experimental results show that our approach can improve the state-of-the-art SMT and CHC solvers, solving considerably more (around 25%) proof tasks involving inductive definitions, demonstrating its efficacy.

1 Introduction

Inductive definitions are prevalent in program verification. Typically, they exhibit in two forms, i.e., algebraic data types (ADTs) and recursively defined functions (RDFs). An ADT defines a type whose elements are constructed by a finite number of applications of a given set of rules. As an example, Listing 1.1 defines natural numbers (`Nat`) as an ADT, and two operations on natural numbers as RDFs. `Nat` is defined as either zero (`zero`) or the successor of another `Nat`, given by `succ(n)`. Both RDFs `plus` and `mult` are specified through assertions with universal quantifiers, serving as axioms.

– The addition operation (`plus`) (lines 2–5) is recursively defined by two cases: (i) adding `zero` to any natural number `y` yields `y`, and (ii) adding the successor of any natural number `x` to `y` yields the successor of `plus(x,y)`.

```
1  datatypes Nat := zero | succ(n: Nat)
2  fun plus(Nat, Nat): Nat {
3    ∀y∈Nat.plus(zero,y) = y
4    ∀x∈Nat,y∈Nat.plus(succ(x),y) = succ(plus(x,y))
5  }
6  fun mult(Nat, Nat): Nat {
7    ∀y∈Nat.mult(zero,y) = zero
8    ∀x∈Nat,y∈Nat.mult(succ(x),y) = plus(mult(x,y),y)
9  }
10 property: ∀x∈Nat,y∈Nat.mult(x,y) = mult(y,x)
```

Listing 1.1: A pseudocode program: the running example

– The multiplication operation (`mult`) (lines 6–9) is recursively defined by two cases as well: (i) multiplying `zero` by any natural number `y` results in `zero`, and (ii) multiplying the successor of any natural number `x` by `y` is the addition of `mult(x,y)` and `y`.

In this example, the proof goal is the commutativity of the multiplication operation `mult`, formulated as `property` at line 10 in Listing 1.1.

Such ADTs are ubiquitous in functional programming languages (e.g., Gallina) for interactive theorem provers (ITPs, e.g., Rocq [3]), and are increasingly being supported by other languages such as TypeScript and Rust.

Inductive Reasoning. Program verification requires reasoning about properties of inductive definitions, including ADTs and RDFs. The standard reasoning process typically consists of two steps: first proving the base case(s) and then proving the inductive case(s) with inductive hypotheses as premises. For example, in Listing 1.1, one may apply induction on the natural number `x`, leading to the base case where `x` is `zero` and an inductive case where `x` is `succ(x')` in which the property is assumed to hold for `x'`. These two cases yield two verification conditions (VCs):

$$\begin{aligned} &\forall\ y \in \texttt{Nat}.\ \texttt{mult}(\texttt{zero}, y) = \texttt{mult}(y, \texttt{zero}), \\ &\forall\ x', y \in \texttt{Nat}.\ \texttt{mult}(x', y) = \texttt{mult}(y, x') \rightarrow \\ &\qquad\qquad\qquad \texttt{mult}(\texttt{succ}(x'), y) = \texttt{mult}(y, \texttt{succ}(x')). \end{aligned} \tag{1}$$

Different techniques can be applied for inductive reasoning, ranging from manual to fully automated. In general, ITPs such as Rocq [3], Isabelle [34] and Lean [32], and semi-automated verifiers (aka deductive verifiers) such as Dafny [28], Stainless [19] and Verus [27], require user intervention.

In contrast, automated constraint solving, including SMT solving and first-order logic theorem proving, is designed to operate without user intervention. Typically, induction schemas are incorporated into SMT solvers [40] or into superposition-based first-order logic theorem provers [8,38,13,16,17,20,18], enabling automatic generation of base and inductive cases as verification conditions with inductive hypotheses as premises. However, many proof goals with inductive definitions cannot be solved by solely applying the inductive hypotheses and axioms. Consequently, additional *auxiliary lemmas* are required to complete the proof. Namely, to prove $A \to P$ where A and P are the premise and the conclusion, respectively, we may introduce a set of auxiliary lemmas $\{L_i\}_{i \in I}$ such that $A \to L_i$ holds for each $i \in I$, and $A \wedge \bigwedge_{i \in I} L_i \to P$ holds.

Lemma Generation Methods and Limitations. This paper tackles *automated generation of auxiliary lemmas* in constraint solving. The existing methods can roughly be classified into three main categories. First, *theory exploration* method, implemented in solvers such as cvc5 [2], iteratively enumerates conjectures starting from small terms and applies heuristic checks to quickly filter out invalid ones before verifying their validity and usefulness for assisting in proving the goal [7,40,42]. Second, the *generalization* method, implemented in solvers such as Vampire [25], identifies common sub-terms in the proof goal and replaces them with new variables to find auxiliary lemmas that are simpler and can generalize the goal [16]. Third, *Constrained Horn Clauses (CHCs) based* method [22] treats inductive definitions as transition systems and solves the verification problem by synthesizing inductive invariants. (Cf. Section 5 for a brief review of the related work.)

Despite their effectiveness in certain scenarios, these methods have significant limitations. Theory exploration is effective for generating simple lemmas but often struggles with discovering more complicated lemmas required for inductive proofs. Indeed, the representative state-of-the-art SMT solver, cvc5, fails to prove the `property` given in Listing 1.1 (cf. the full version of the paper for details). Generalization methods have limited expressiveness and cannot handle more general problems. CHC-based methods have limited capability to handle inductive definitions, particularly RDFs. Overall, these traditional logic-based methods mostly rely on fixed mechanisms and heuristic strategies, which often suffer from limited scalability or expressiveness, restricting the usability and applicability of automated constraint solving in program verification.

LLM-aided Solving. LLMs have demonstrated remarkable capabilities in code generation [15,46,45,52], program specification inference [49,31], loop-invariant generation [5,50,36], and interactive theorem proving [39,48]. Recent works have also explored LLM-assisted lemma/conjecture generation in deductive verification (e.g., Dafny [41] and Verus [51]), hardware model checking [35] and mathematical reasoning [1,47,6]. Overall, these methods largely follow a generate-then-verify paradigm, but they target either program-level proof construction or hardware/mathematical formula reasoning, rather than fully automated inductive reasoning and first-order logic constraint solving.

Inspired by these advances of LLMs in generative AI and formal reasoning, we introduce LLMs into automated inductive reasoning, in particular, to overcome the limitations of pure logic-based methods in lemma generation. However, there are two main technical challenges:

Challenge 1. While LLMs often exhibit emergent abilities across diverse problem types, they may not be very effective in specific downstream tasks. The primary challenge is to effectively unleash their capability for inductive reasoning, e.g., to guide them to recognize what constitutes *good* auxiliary lemmas therein.

Challenge 2. Unlike traditional logic-based methods, the outputs of LLMs exhibit randomness, and can even hallucinate. The second challenge is how to quickly eliminate invalid outputs and ensure their validity and usefulness for assisting in proving the target goal.

Our Contributions. We propose a neuro-symbolic approach that synergistically integrates LLMs and constraint solving for automated lemma generation in inductive reasoning. Our approach consists of three stages: *query*, *filter* and *validate*. Given an instance of the inductive reasoning problem that comprises inductive definitions and a proof goal, our approach automatically generates auxiliary lemmas and proves the goal. The three stages are designed to address the above two technical challenges. To address Challenge 1, we design two prompt strategies in the query stage: (i) the first strategy imitates human inductive reasoning through equational term rewriting, enabling goal-oriented lemma discovery, and (ii) the second strategy abstracts the proof goal into simpler forms and generates bridging lemmas between simplified forms and the original goal. Moreover, it incorporates heuristics, such as suggesting commonly used axioms, attempting to generate strengthened propositions that imply the original goal, and applying equivalent transformations to terms in the proof goal. The filter and validate stages together address Challenge 2, where the filter stage quickly eliminates incorrect and useless outputs using backend solvers with short timeouts, resulting in candidate lemmas, while the validate stage enforces validity and usefulness of candidate lemmas through a two-step process: checking whether the candidate lemmas can assist in proving the original goal, and then recursively verifying each candidate lemma if they suffice to prove the goal.

We evaluate our approach on 706 instances of the inductive reasoning problem collected from diverse benchmarks commonly used in the inductive reasoning literature [40,22,44]. We compare our method against state-of-the-art solvers (including cvc5 and Vampire). The experimental results show that our approach achieves (at least 25%) higher success rate than these solvers. Additionally, the ablation studies show that both the filtering and prompt designs contribute substantially to the performance improvement. Further experiments also suggest that our approach is robust across different LLMs, sampling temperatures and backend solvers, which demonstrates the generality of our approach.

The full version of the paper, as well as the tool, benchmarks and experimental data, is available at `https://github.com/fengwz17/LLM4Ind`.

2 Preliminary

2.1 Satisfiability Modulo Theories

Satisfiability Modulo Theory (SMT) refers to deciding whether a given logical formula is satisfiable, i.e., whether there exists an assignment of its variables over background theories, under which the formula holds.

Formally, an SMT formula is defined over a given signature Σ that consists of a set of sorts (types) $\mathcal{S}$, function symbols $\mathcal{F}$ and predicate symbols $\mathcal{P}$. A model $\mathcal{M}$ of a given theory $\mathcal{T}$ (e.g., Linear integer arithmetic (LIA), Bit-vector (BV), Uninterpreted functions (UFs), Algebraic data types (ADTs)) provides an interpretation of sorts, function symbols and predicate symbols in Σ. A set of clauses $F = \{C_1, \cdots, C_n\}$ represents the conjunction $C_1 \wedge \cdots \wedge C_n$, where each clause C_i may be universally quantified.

SMT solvers, such as Z3 [33], cvc5 [2] or Yices [12], combine SAT-solving techniques with specialized theory solvers. The SAT engine handles the propositional structure of the formula, while theory solvers reason about constraints in their respective domains. Through this integration, SMT solvers have become powerful tools for, among others, program verification and synthesis, model checking, and test-case generation. Besides, some first-order logic provers, such as Vampire [25], can also accept a fragment of SMTLIB2 inputs [37]. For simplicity, we refer to both SMT solvers and such first-order logic provers as *solvers* in this paper when no ambiguity arises.

2.2 Inductive Definitions

Algebraic Data Types. The theory of algebraic data types (ADTs) is defined over a signature Σ. For example, natural numbers and lists can be defined as:

```
datatypes Nat := zero | succ(n: Nat),
datatypes List := nil | cons(head: Nat, tail: List).
```

Σ includes sorts for representing ADTs (e.g., `Nat`, `List`), function symbols for constructors (e.g., `zero`, `succ` for `Nat`) and selectors (e.g., `head`, `tail` for `List`), and predicate symbols for testers.

Recursively Defined Functions. A recursively defined function (RDF) is a function whose definition refers to itself on smaller or simpler inputs. Formally, such a function is specified by a set of base cases, which provide values for the simplest inputs, together with recursive rules that reduce larger or more complex inputs to instances of the same function on smaller arguments. This structure ensures that every call to the function eventually reaches a base case, guaranteeing well-definedness and termination. RDFs are fundamental in mathematics and computer science, as they naturally capture inductive structures such as sequences, trees and lists.

2.3 Lemma Generation

Let A denote a set of axioms encoding RDFs over ADTs, and P denote a target property to be proved. A usually consists of universally quantified equations, while P is a universally quantified formula, e.g., $\forall \vec{x} \in \tau.\ t(\vec{x})$, where $\vec{x} = \{x_1, \cdots, x_n\}$, τ is an ADT sort (e.g., `Nat`), and t is a term over $\vec{x}$. The goal is to establish the validity of $A \rightarrow P$. To this end, we aim to generate a set of lemmas $\mathcal{L} = \{L_i\}_{i \in I}$ such that the following conditions hold:

$$A \wedge \bigwedge_{i \in I} L_i \rightarrow P, \quad \text{and} \quad \forall i \in I.\ \ A \rightarrow L_i.$$

In practice, to prove $A \rightarrow P$, SMT-based approaches negate the property P, and check whether the formula, given by the set of clauses $F = \{A, \neg P\}$, is unsatisfiable. Then the verification task is checking the satisfiability of F. The *lemma generation task* is to find a set of lemmas $\mathcal{L} = \{L_i\}_{i \in I}$ such that: (i) each lemma L_i is entailed by A, i.e., $\{A, \neg L_i\}$ is unsatisfiable, and (ii) the set of clauses $\{A, \neg P\} \cup \mathcal{L}$ is unsatisfiable.

SMT solving commonly copes with universally quantified formulas such as $\forall \vec{x} \in \tau.t(\vec{x})$ through instantiation-based techniques, and handles the negation of universally quantified formulas (effectively existentially quantified formulas since $\neg\forall \vec{x} \in \tau.t(\vec{x})$ is equivalent to $\exists \vec{x} \in \tau.\neg t(\vec{x})$) via skolemization. cvc5 implements an inductive strengthening technique in its skolemization module to incorporate the induction schema into SMT solving and a theory exploration method to automatically generate auxiliary lemmas. There are other solvers such as Vampire [18] and Racer [22] which support reasoning about inductive definitions and are reported to outperform cvc5 in some categories of benchmarks. However, in general, cvc5 represents the state-of-the-art in inductive reasoning, which is used as the backend solver.

Hereafter, we adopt the following terminology for the LLM-aided lemma generation task. Outputs generated by LLMs are referred to as *conjectures* if they are in the correct format. Conjectures $\{L_i\}_{i \in I}$ are said *useful* if they can assist in proving P when used as premises, namely, $A \wedge \bigwedge_{i \in I} L_i \rightarrow P$ can be proved. A conjecture L is referred to as a *lemma* if it can be proved under the axiom A. Useful conjectures $\{L_i\}_{i \in I}$ are referred to as *auxiliary lemmas* when all of them can be proved under the axiom A, i.e., $A \rightarrow L_i$ can be proved for all $i \in I$. To establish the validity of $A \rightarrow P$, our main goal is to elicit LLMs to generate auxiliary lemmas.

3 Our Neuro-Symbolic Approach

3.1 Motivating Example

We first illustrate the challenges of naively querying LLMs for lemma generation without carefully designed prompts. Consider the task of verifying the property in Listing 1.1 using the following naive prompt:

```
You are an expert in constraint solving, inductive
    reasoning, and functional program verification.
Please generate auxiliary lemmas in SMTLIB2 format to
    add to the following file, which can help the solver
     verify the property.

  { Input SMTLIB2 file }
```

Listing 1.2: A naive prompt for lemma generation

Due to randomness and possible hallucination, LLM-generated conjectures may not be auxiliary lemmas. Based on our experiment, these conjectures can be categorized into the following three types:

- **Incorrect Conjectures.** When the outputs of LLMs are complete and syntactically correct, forming conjectures, they may be semantically incorrect. Namely, an LLM-generated conjecture L contradicts the axiom A, making the premise $A \wedge L$ false. In this case, while $A \wedge L \rightarrow P$ is vacuously true, the conjecture L does not hold under the axiom A and is useless. For instance, LLMs may generate the following conjecture for the motivating example:

$$\forall\ x \in \texttt{Nat}.\ \texttt{plus}(x, \texttt{zero}) = \texttt{zero}$$

  which contradicts the axiom that $\texttt{plus}(x, \texttt{zero}) = x$.
- **Correct but Useless Conjectures.** Even when an LLM-generated conjecture L is correct under the axiom A, i.e., $A \rightarrow L$ holds, it may still be useless for proving the property P. For instance, an LLM generates a conjecture L that is identical to the property P or a generic property that is completely irrelevant to the property P. For instance, LLMs may generate the following conjectures for the motivating example:

$$\forall\ x, y, z \in \texttt{Nat}.\ \texttt{plus}(\texttt{plus}(x, y), z) = \texttt{plus}(x, \texttt{plus}(y, z)),$$
$$\forall\ x, y \in \texttt{Nat}.\ \texttt{plus}(x, y) = \texttt{plus}(y, x), \qquad \forall\ x \in \texttt{Nat}.\ \texttt{mult}(x, \texttt{zero}) = \texttt{zero}).$$

  These conjectures, while valid, provide no assistance in proving P.
- **Useful but Remaining to be Proved Conjectures.** While LLMs can generate useful conjectures for proving the property P, these conjectures themselves should be proved from the axiom A, to qualify as auxiliary lemmas. In practice, proving useful conjectures themselves is non-trivial, e.g., some cannot be directly proved via SMT solving, leaving them as new proof obligations. For instance, for the motivating example, LLMs may generate the following useful conjecture:

$$\forall\ x, y \in \texttt{Nat}.\ \texttt{mult}(x, \texttt{succ}(y)) = \texttt{plus}(\texttt{mult}(x, y), x).$$

  This conjecture cannot be directly proved via SMT solving using cvc5, and thus requires further verification with additional auxiliary lemmas.

This example demonstrates that state-of-the-art LLMs have the potential to understand inductive reasoning and generate useful conjectures beyond existing fixed mechanisms and heuristic strategies. (Note that this example cannot be proved by the representative state-of-the-art solver cvc5; cf. the full version of the paper.) However, LLMs still lack clear reasoning strategies and may generate conjectures which are incorrect, useless, or themselves non-trivial to prove. Effectively leveraging LLMs requires carefully designed prompts to guide lemma generation and a systematic workflow to validate and integrate these conjectures, which motivates our approach.

3.2 Workflow

To bring the best of LLM-aided solving and traditional logic-based solving, we propose a neuro-symbolic approach, which synergistically integrates LLMs and

Algorithm 1: Main Workflow

```
max_depth ← 3, max_iter_number ← 3; // In default
// Input: labeled SMTLIB2 file P and current recursion depth d;
// Output: True if proved, False otherwise;
Function ProveRun(P, d):
    if initialCheck(P) then
        return True;
    if d ≥ max_depth then  return False ;
    foreach prompt ∈ prompt_pool do
        for i ← 1 to max_iter_number do
            r, subgoals ← Prove(P, prompt);
            if r = True then
                all_success ← True;
                parallel for subgoal ∈ subgoals do
                    if ¬ProveRun(subgoal, d + 1) then
                        all_success ← False;
                        break;
                if all_success then
                    return True;
    return False;
```

SMT solvers to iteratively generate conjectures, check their validity and usefulness for assisting in proving the target property. After preprocessing the input file in SMTLIB2 format, we invoke the `ProveRun` function as the main workflow to prove the property.

Preprocessing. Given an instance of the inductive reasoning problem as an SMTLIB2 file, we parse the input file and organize it as three parts: datatype definitions, recursive function definitions, and the verification target, each of which is labeled with a comment (e.g., `; datatype definitions`, `; function definitions`, and `; proof goal`) to help LLMs' understanding and reasoning.

The `ProveRun` function. The main workflow is implemented in the `ProveRun` function in Algorithm 1. It takes a labeled SMTLIB2 file P and the current recursion depth d (initialized to 0) as input, attempts to prove P (for clarity, the property specified in P is also referred to as P). The recursive execution of `ProveRun` forms a proof tree whose depth and width are bounded by the configurable parameters max_depth and $max\_iter\_number \times |\texttt{prompt\_pool}|$, where max_depth bounds the depth of recursive calls, max_iter_number bounds the number of queries to LLMs for each prompt, and `prompt_pool` comprises the designated prompts.

In detail, `ProveRun` first checks whether P can be *directly* proved via SMT solving, without LLM-generated conjectures. If it cannot be proved and the recursion has not yet reached the depth bound max_depth, `ProveRun` attempts to prove P using designated prompts from the prompt pool through three key stages *query*, *filter* and *validate*.

Algorithm 2: The Prove Function

```
  // Input: labeled SMTLIB2 file P, current prompt prompt;
  // Output: (True,C) if P is proved with useful conjectures C;
  //         (False, ∅) if conjectures cannot assist in proving P;
1 Function Prove(P, prompt):
2     C ← LLMQuery(P, prompt);
3     foreach c ∈ C do
4         if isFiltered(c) then
5             return (False, ∅);
6     if Verify(P, C) then
7         return (True, C);
8     return (False, ∅);
```

Each prompt is queried up to max_iter_number times (three times in this work, considering the cost and effectiveness), at lines 6–16. In this work, we design two effective prompt strategies (cf. Section 3.3) to form the prompt pool, while other new prompt strategies can be easily extended into our workflow.

For each query of a prompt `prompt`, `ProveRun` invokes the function `Prove` (line 8), which attempts to prove P assisted by conjectures generated by LLMs using `prompt`. If P is proved by `Prove` using `prompt`, with LLM-generated conjectures $\{L_i\}_{i \in I}$, i.e., $A \wedge \bigwedge_{i \in I} L_i \rightarrow P$ is proved, then these conjectures $\{L_i\}_{i \in I}$ are regarded as sub-goals and are validated by recursively invoking `ProveRun` in parallel. We can complete the proof of P if all the sub-goals can be proved, i.e., $A \rightarrow L_i$ for $i \in I$ are proved.

The `Prove` function. The `Prove` function (Algorithm 2) first queries the specified LLM using `prompt`, which will produce a set of conjectures C. The LLM-generated conjectures are iteratively checked by invoking `isFiltered` (line 4), which is designed to quickly identify incorrect and useless conjectures. If none of these conjectures can be filtered out, `Prove` invokes the `Verify` function (line 6) which checks whether these conjectures suffice to prove P (line 6), i.e., checking the unsatisfiability of $\{A, \neg P\} \cup C$. If this is the case (line 7), these conjectures are returned, which will be validated as sub-goals by recursively invoking `ProveRun`. Otherwise, `Prove` returns `False`.

The `isFiltered` function. The filtering stage, implemented via `isFiltered`, is designed to quickly identify incorrect and useless conjectures. An LLM-generated conjecture c is filtered out by `isFiltered`, i.e., `isFiltered`(c) returns `True`, if (i) c has syntactic errors checked by the parser of an SMT solver (e.g., cvc5), or (ii) c is identical to the proof goal P compared by syntactic equivalence checking, or (iii) c is inconsistent with the axiom A, i.e., $A \wedge c$ is unsatisfiable, via SMT solving. We check the inconsistency of A and c instead of the invalidity of $A \rightarrow c$ (equivalently, the satisfiability of $A \wedge \neg c$) because: $A \wedge c$ usually contains only the universal quantifier $\forall$ while $A \wedge \neg c$ contains both the universal quantifier $\forall$ and existential quantifier $\exists$, thus $A \wedge c$ is easier to be checked; moreover the unsatisfiability of $A \wedge c$ entails the satisfiability of $A \wedge \neg c$, thus c cannot be a lemma under the axiom A.

The `Verify` function. The validation stage implemented via `Verify` is used to check whether the LLM-generated conjectures C are *useful* for proving the goal. It is done by checking the unsatisfiability of $\{A, \neg P\} \cup C$, which is equivalent to checking $A \wedge \bigwedge_{c \in C} c \rightarrow P$. If the unsatisfiability of $\{A, \neg P\} \cup C$ can be proved by SMT solving, `Verify` returns `True`, otherwise `False`.

3.3 Prompt Strategies for Lemma Generation

To effectively elicit LLMs to generate useful lemmas, we design two prompt strategies, each of which consists of three parts: `Task Description` (explaining the SMTLIB2 format and the task), `Chain of Thoughts` (providing the reasoning strategy), and `Output Format` (specifying how to format the output).

Strategy 1: Equational Reasoning. This strategy guides LLMs to perform step-by-step, human-like equational reasoning. Concretely, for a proof goal of the form $\forall x \forall \vec{y}.\ f(x, \vec{y}) = g(x, \vec{y})$, where f is inductively defined over the variable x, the prompt instructs the LLM to: (i) identify the inductive definition of f from the labeled SMTLIB2 file; (ii) determine whether the base case requires auxiliary lemmas and, if it can be proved by the solver's build-in simple induction, proceed directly to the inductive case; (iii) for the inductive case $\forall x' \forall \vec{y}.\ f(t(x'), \vec{y}) = g(t(x'), \vec{y})$, where x' is the variable used in the inductive hypothesis (e.g., $\forall x' \forall y.\ \texttt{mult}(\texttt{succ}(x'), y) = \texttt{mult}(y, \texttt{succ}(x'))$), transform the left-hand side $f(t(x'), \vec{y})$ step-by-step using known premises (axioms and the inductive hypothesis); and (iv) when a step cannot be derived directly from known premises, generate it as a conjecture.

Example 1. To prove $\forall x, y \in \texttt{Nat}.\ \texttt{mult}(x, y) = \texttt{mult}(y, x)$ (i.e., the property in Listing 1.1), using Strategy 1, an LLM may reason as:

```
consider the inductive case:
      mult(succ(x'), y)) =:| axiom of mult |:
    plus(mult(x', y), y) =:| inductive hypothesis |:
    plus(mult(y, x'), y) =:| unknown conjecture |:
      mult(y, succ(x'))
```

The last highlighted step cannot be directly derived from known premises, so the following conjecture is generated by the LLM:

$$\forall\ x, y \in \texttt{Nat}.\ \texttt{plus}(\texttt{mult}(y, x), y) = \texttt{mult}(y, \texttt{succ}(x)).$$

Though this conjecture suffices to prove the property, it cannot be directly proved via SMT solving (e.g., cvc5 with 1200s time limit). Thus, it will be regarded as a sub-goal on which Strategy 1 is applied again, finally resulting in the following useful conjecture:

$$\begin{aligned} \forall\ x, y \in \texttt{Nat}.\ &\texttt{plus}(\texttt{plus}(\texttt{mult}(y, x), x), \texttt{succ}(y)) = \\ &\texttt{plus}(\texttt{plus}(\texttt{mult}(y, x), y), \texttt{succ}(x)). \end{aligned} \quad (2)$$

Strategy 2: Term Rewriting and Generalization. Unlike Strategy 1, this strategy does not instruct LLMs to think step-by-step, but provides common ideas in lemma generation, encouraging them to generate lemmas more flexibly and simplify the proof goal P. Concretely, the prompt instructs the LLM to: (i) generate basic axioms; (ii) strengthen the conclusion, i.e., find a stronger lemma that is easier to prove; (iii) identify a common term that appears on both sides of the proof goal P and replace it with a fresh variable, leading to a simplified proof goal P'; (iv) if no common terms exist, attempt to rewrite terms in P using function definitions to have a common term; and (v) if necessary (e.g., P' is not enough to prove P), generate bridging lemmas between P' and P as conjectures such as $P' \to L$ and $L \to P$.

Example 2. Suppose the LLM fails to generate useful lemmas for proving the sub-goal in Equation (2), via Strategy 1. Then, Strategy 2 is used to guide the LLM, which identifies the common term $\texttt{mult}(y, x)$, rewrites the sub-goal, and generates the following simplified sub-goal:

$$\forall\ t, x, y \in \texttt{Nat}.\ \texttt{plus}(\texttt{plus}(t, x), \texttt{succ}(y)) = \texttt{plus}(\texttt{plus}(t, y), \texttt{succ}(x)).$$

This simplified sub-goal can be directly proved via SMT solving, thus we can complete the overall proof of the property. A further example of Strategy 2 is given in the full version of the paper.

4 Evaluation

We implement our approach in a tool, named LLM4Ind. We evaluate LLM4Ind to answer the following research questions (**RQs**):

RQ1. How effective is our approach at solving constraints involving inductive definitions compared with state-of-the-art solvers?
RQ2. What is the contribution of our prompt and filtering designs to the overall performance?
RQ3. Is our approach robust over different LLM models and sampling temperatures that controls creativity and diversity of the output?

Benchmarks. We collected four inductive reasoning benchmarks from prior work: StandardDT, StandardDTLIA, Autoproof, and IndBen, which are commonly used in literature [40,44,26,10,22].

- StandardDT [40] comprises 241 proof tasks in the ADT theory which was collected from test suites of IsaPlanner [11], CLAM [21] and HipSpec [7], and verification conditions in functional program verification in Leon [4]. It is often regarded as a standard benchmark for automated inductive reasoning.
- StandardDTLIA [10] contains 168 proof tasks which were derived from StandardDT by converting the `Nat` type and all the related operations to `Int` in the LIA theory, covering both ADT and LIA theories.
- AutoproofBM [44] comprises 141 proof tasks, which use more complex ADT and LIA definitions and are relatively more challenging to solve.

– IndBen [16] provides 3,396 proof tasks among which we select 156 representative tasks, one task per group, as the tasks in the same group define the same definitions and property (differing only in syntax). The 156 proof tasks cover common ADT theories such as `List`, `Nat` and `Tree`.

Overall, we collected 706 proof tasks as our subject.

Experimental Setup. To answer **RQ1**, we compare LLM4Ind with cvc5, Vampire and Racer in terms of the number of solved proof tasks.

– cvc5 [2] is one of the leading SMT solvers. As its performance may vary with options and versions [44,26], we run cvc5 (with three option configurations) and cvc4 (with the default inductive reasoning setting) in parallel (cf. the full version of the paper for detail options), which serves as the baseline for cvc5 and acts as the backend SMT solver in our workflow (for clarity, it is referred to directly as cvc5).
– Vampire [25,18] is a leading automated first-order logic prover based on superposition calculus. It supports a fragment of the SMTLIB2 inputs. We run Vampire under the portfolio mode, which runs multiple proving strategies in parallel.
– Racer [22] extends the CHC solver Spacer [24] to better support ADTs and RDFs. As it only supports inputs in CHCs (not general SMTLIB2) form, we only compare with it using its provided benchmarks.

To answer **RQ2**, we conduct an ablation study. To answer **RQ3**, we evaluate LLM4Ind using four different LLMs: DeepSeek-v3.2 (DeepSeek), Qwen3-235B-instruct (Qwen), Gemini 2.5 flash (Gemini) and GPT-5 and vary the sampling temperature. Note that we use Qwen as the default LLM unless stated explicitly otherwise, as a trade-off between the cost and performance.[5]

All experiments were conducted on a machine with an Intel(R) Xeon(R) Platinum 8368Q CPU (76 cores, 2.60 GHz), 1024 GiB RAM and Ubuntu 22.04.5 LTS (1024 GiB RAM is the memory for evaluation instead of consumption). The solving of each proof task is limited to 1200 seconds (wall time) unless stated explicitly otherwise. For LLM4Ind, we use both prompt strategies described in Section 3.3, each invoked for up to 3 iterations as in Algorithm 1 (line 6 and line 7). The backend solver is given 60 seconds for initial checking (Algorithm 1, line 3) and for each invocation of the `Verify` function to verify candidate conjectures (Algorithm 2, line 6). The filtering checking (Algorithm 2, line 4) is given 1 second per conjecture. At most 4 threads are used to verify subgoals concurrently (Algorithm 1, line 11). We set the hyperparameter top_p=0.9 and sampling temperature=0.9 by default according to [29]. For cvc5, we run 4 threads in parallel, one thread per option configuration. Other tools use their default thread configurations without additional parallelization settings.

[5] The price per M input/output tokens: DeepSeek-v3.2 ($0.27/$0.40), Qwen3-235B ($0.072/$0.464), Gemini 2.5 flash ($0.30/$2.50), GPT-5 ($1.25/$10.00).

Table 1: Comparison with baseline SMT solvers

Benchmark	Total	LLM4Ind		cvc5		Vampire		Racer	
		<1200s	<360s	<1200s	<360s	<1200s	<360s	<1200s	<360s
StandardDT	241	**212**	**200**	150	150	160	160	N/A	N/A
StandardDTLIA	168	**134**	**127**	76	76	58	56	40	40
AutoProofBM	141	**65**	**61**	34	34	3	3	N/A	N/A
IndBen	156	114	104	34	34	**122**	**118**	N/A	N/A
Total	706	**525**	**492**	293	293	343	337	40	40
Avg time (s)		97.30	58.66	**3.38**	**3.38**	19.39	9.26	6.09	6.09

4.1 RQ1. Effectiveness

The results are reported in Table 1, including the number of tasks solved by each tool with the time limits 1200s and 360s, as well as the average solving time of the solved tasks. Recall that Racer cannot directly verify StandardDT, AutoProofBM and IndBen (hence is marked by N/A). For reference, a full run of LLM4Ind over all benchmarks consumed approximately 11.5M tokens using Qwen, corresponding to an estimated cost of about $4.

Overall, LLM4Ind solved significantly more tasks than the baselines, regardless of the time limit. For instance, under 1200s, LLM4Ind solved 232 more tasks than cvc5 and 182 more tasks than Vampire. This confirms the effectiveness of our approach. On StandardDTLIA, LLM4Ind also solved 94 more tasks than Racer. On IndBen, Vampire solved more tasks than LLM4Ind, but we will see later in Table 2 that LLM4Ind surpasses Vampire when more powerful LLMs (Gemini and GPT-5) are used instead of Qwen.

It is not surprising that LLM4Ind's solving time is higher than the traditional logic-based approaches, because it recursively queries the LLM and invokes the SMT solver. Arguably, the time remains acceptable for verification tasks (on average about 100s), especially when producing a correct result is more critical.

We note that the recent tools AutoProof [44] and CCLemma [26] have reported promising results on lemma generation for inductive reasoning. However, these tools support neither SMTLIB2 nor CHC format inputs. As an *indirect* comparison using the results reported in [44], AutoProof solved 161 tasks while LLM4Ind solved 180 tasks using their benchmarks.

4.2 RQ2. Ablation Study

To study the contribution of our prompt and filtering designs, we compare our prompt strategies with the naive prompt strategy (cf. Listing 1.2) and evaluate LLM4Ind with and without the filter.

Prompt Design. The results are reported in Table 2 using four different LLMs: Qwen, DeepSeek, Gemini and GPT-5. Here, LLM4Ind uses the prompt strategies presented in Section 3.3, while the naive one (cf. Listing 1.2) provides only a simple task environment description, requests lemma generation, and specifies output format. For fairness, the naive prompt is invoked up to 6 times, matching

Table 2: Results of LLM4Ind using different LLMs and naive prompt strategy

Benchmark	Qwen		DeepSeek		Gemini		GPT-5	
	LLM4Ind	Naive	LLM4Ind	Naive	LLM4Ind	Naive	LLM4Ind	Naive
StandardDT	**212**	160	**206**	153	**206**	155	**212**	180
StandardDTLIA	**134**	88	**131**	90	**119**	91	**144**	106
AutoProofBM	**65**	42	**63**	43	**68**	49	**67**	46
IndBen	**114**	89	**121**	79	**126**	76	**129**	118
Total	**525**	379	**521**	365	**514**	371	**552**	450

Table 3: Results of LLM4Ind with and without filter over three runs

Benchmark	W. filter				W.o. filter				Avg time (s)	
	R1	R2	R3	Avg	R1	R2	R3	Avg	W. filter	W.o. filter
StandardDT	212	211	207	**210**	204	203	209	205.3	83.91	**67.26**
StandardDTLIA	134	137	135	135.3	134	139	134	**135.7**	**97.53**	112.05
AutoProofBM	65	61	64	**63.3**	60	64	64	62.7	**99.10**	103.61
IndBen	114	111	111	**112**	113	106	111	110	**168.92**	178.64
Total	525	520	517	**520.7**	511	512	518	513.7	107.59	**107.37**
Tokens (M)	11.50	12.08	11.57	35.15	13.37	11.75	12.49	37.61	N/A	N/A

the others (2 prompts$\times$3 times/prompt). LLM4Ind with our prompt strategies significantly outperforms the naive one, across all LLM models and benchmarks.

Filtering design. The results are reported in Table 3. Because solving outcomes depend heavily on the quality of LLM outputs, we run each setting (with and without the filter) three times (R1, R2 and R3) and report the average results (Avg). For a fair comparison, the average time is computed over all solved tasks. For per-task time comparison across all four benchmarks, we provide scatter plots in the full version of the paper, visualizing the solving time differences for each proof task.

In general, with the filter, LLM4Ind solves more tasks (520.7 vs. 513.7 in total over three runs), indicating that the filtering design is effective. In many cases, without the filter (w.o. filter), incorrect or useless conjectures consume substantial solving time and can lead to time out. In contrast, with the filter (w. filter), some iterations are discarded early, saving time and allowing those instances to be solved successfully.

In detail, while the filter can reduce the solving time on the three benchmarks (i.e., StandardDTLIA, AutoProofBM and IndBen), StandardDT remains an exception. It may be because StandardDT contains relatively simple ADT definitions (primarily common types such as `List`, `Nat`, and `Tree`) for which LLMs can generate correct conjectures quickly, so that filtering provides little benefit but incurs additional overhead. In addition, the last row of Table 3 reports the total tokens (in millions) consumed across the six runs. Overall, enabling the filter results in lower token usage compared to the configuration without the filter, indicating improved cost efficiency.

4.3 RQ3. Robustness of LLM4Ind

To evaluate the robustness of LLM4Ind across different LLMs, we consider DeepSeek, Gemini and GPT-5, besides Qwen. The experimental results are pre-

Table 4: Results of LLM4Ind with different sampling temperatures

Benchmark	T = 0.1			T = 0.5			T = 0.9			T = 1.3		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
StandardDT	211	206	210	207	209	207	**212**	211	207	208	209	210
StandardDTLIA	135	139	135	136	138	138	134	137	135	**140**	136	135
AutoProof	60	62	62	61	62	61	**65**	61	64	60	60	**65**
IndBen	114	108	114	110	107	113	114	111	111	**119**	113	112
Total	520	515	521	514	516	519	525	520	517	**527**	518	522
Range Std	6 3.2			5 2.6			8 4.2			9 4.7		

Table 5: Comparison with baseline SMT solvers (Vampire backend)

Benchmark	Total	LLM4Ind-V		cvc5		Vampire		Racer	
		<1200s	<360s	<1200s	<360s	<1200s	<360s	<1200s	<360s
StandardDT	241	**214**	**206**	150	150	160	160	N/A	N/A
StandardDTLIA	168	**80**	73	76	**76**	58	56	40	40
AutoProofBM	141	19	16	**34**	**34**	3	3	N/A	N/A
IndBen	156	**140**	**140**	34	34	122	118	N/A	N/A
Total	706	**453**	**435**	293	293	343	337	40	40
Avg time (s)		59.89	38.74	**3.38**	**3.38**	19.39	9.26	6.09	6.09

sented in Table 2, from which we observe that LLM4Ind consistently improves cvc5, Vampire and Racer, regardless of the LLM model used. Moreover, to assess robustness under different samplings, we set the temperature of Qwen to 0.1, 0.5, 0.9 and 1.3, respectively. We ran three independent runs for each temperature to account for the randomness of LLM outputs. Table 4 reports the per-run counts. For each temperature, we also report the range (max−min over the three runs) and the standard deviation (std) of the total number of solved tasks to quantify variability. The range is at most 9 (e.g., 518–527 for $T = 1.3$), and the std remains below 5 in all cases, indicating low variance across runs. We observe that as the temperature increases, the variability tends to increase slightly (e.g., the std grows from 3.2 at $T = 0.1$ to 4.7 at $T = 1.3$, and the range increases from 6 to 9), which aligns with the expectation that higher temperatures introduce greater randomness into LLM sampling. In summary, the number of solved tasks varies only slightly across runs and temperatures. These results suggest that the inherent randomness of LLMs and the sampling temperature have a minor impact on LLM4Ind's performance, demonstrating its robustness over different runs and temperatures.

We also replace cvc5 in our workflow with Vampire (denoted as LLM4Ind-V). Table 5 reports the results. Overall, LLM4Ind-V solved more proof tasks than each of the three baselines. Compared with Vampire, LLM4Ind-V shows improvements across all four benchmarks, demonstrating that our approach can enhance Vampire's performance and that LLM4Ind remains robust across different backend solvers. However, compared to cvc5, LLM4Ind-V underperforms in two cases. In particular, for StandardDTLIA mixed with ADT and LIA theories, Vampire has limited support, so LLM4Ind-V solved fewer tasks than cvc5 under the 360s limit. For AutoProofBM, 119 (out of 141) proof tasks involve ADT symbols defined in the SMTLIB2 standard. They are supported by cvc5, but not by Vampire. As a result, although our approach improves Vampire's performance, LLM4IND-V still solved fewer tasks than cvc5 on this benchmark.

5 Related Work

Inductive reasoning is supported, with different degrees of human intervention, by many theorem provers. Existing works address automated inductive reasoning mainly from two perspectives. (1) Incorporating *induction schemas* into decision procedures that originally do not support induction. For instance, [40] enabled SMT solvers to automatically handle inductive definitions by integrating induction schemas into quantifier elimination, while works [8,38,13,17,20,18] extended the superposition calculus with induction rules. (2) Generating auxiliary lemmas to enhance inductive reasoning in provers that support induction, such as automated theorem provers for functional languages, including IsaPlanner [11], ACL2 [23], Zeno [43] and HipSpec [7]. Below, we briefly review representative approaches on automated lemma generation.

Theory Exploration. The main idea is to construct a pool of candidate lemmas from available symbols, typically by term enumeration or instantiating formula templates. The candidates are then filtered using heuristics. HipSpec [7] is a representative tool in this category, which uses counterexample and congruence closure to efficiently filter out invalid lemmas. CCLemma [26] borrows this pruning idea and proposes e-graph guided lemma discovery to make theory exploration more goal-directed. In addition, cvc5 [40] also adopts a similar idea: it enumerates conjectures in increasing size, and filters them using techniques based on active conjectures, equivalence-class construction, and ground-facts reasoning. ADTInd [53] employs syntax-guided templates to enumerate lemmas and implements a generalization mechanism for refining them.

Generalization. Generalization techniques identify common sub-terms in the proof goal and replace them with fresh variables to find auxiliary lemmas that are simpler and can generalize the target property. The main heuristic is to select the common sub-term for replacement by introducing new variables in positions where induction can potentially be applied. This approach has been adopted by modern theorem provers such as ACL2 [23]. Zeno [43] applies the same common sub-term technique but combines it with counterexample searching to avoid over-generalization. Vampire [25,16] introduces an *induction with generalization* inference rule within the superposition calculus framework. This rule can handle properties with multiple occurrences of the same induction term, and instantiates induction axioms with logically stronger variants of the property being proved. AutoProof [44] differs from generalization-based approaches, but can be regarded as a goal-oriented method. It transforms the goal into *induction-friendly forms* that guarantee effective use of inductive hypotheses. Moreover, it synthesizes lemmas as equations where one side matches a term in the goal, enabling systematic rewriting in proof. This directed lemma synthesis avoids enumerating useless lemmas and ensures progress toward provable sub-goals.

CHC-based Method. These approaches treat inductive definitions as transition systems and solve the problem by synthesizing inductive invariants. Several CHC solvers can handle ADTs, but reasoning about RDFs remains challenging due to the undecidability of the underlying logic. For instance, VeriMAP [9,10] transforms CHCs with ADTs to CHCs over basic types, but the transformation is unsound for UNSAT answers. Racer [22] addresses this challenge by approximating RDFs abstractions. It compiles RDFs to CHCs while preserving satisfiability and replaces RDFs with finite unfoldings parameterized by depth k. It implements an IC3-style algorithm, enabling automatic learning of inductive invariants over ADTs and RDFs.

Neuro-Symbolic Method. These approaches largely follow a generate-then-verify paradigm, in which neural models propose candidate lemmas, and symbolic tools are used to validate or refute them. Several works apply this idea in deductive verification or ITPs, where LLMs assist users by proposing lemmas that are subsequently checked by a verifier or proof assistant (e.g., Dafny [41] and Verus [51], and ITPs such as Lean4 [47] and Isabelle [1]). These approaches primarily aim to enhance user productivity or guide interactive proof search, rather than enable fully automated reasoning.

Recent work [35] studies LLM-assisted invariant generation for hardware model checking, where LLMs propose candidate inductive invariants over hardware transition systems. Similarly, [14] learns induction predicates for inductive and arithmetic problems using traditional neural models. It focuses on conjectures over inductive definitions encoded as SMT formulas in integer theory, and is not designed as a general LLM-based framework. AquaForte [30] applies LLMs to generate candidate instantiations for quantified uninterpreted functions in SMT solving, primarily targeting satisfiability checking. In contrast, our work focuses on automated proof generation by establishing the unsatisfiability of the negation of the target property.

6 Conclusion

In this work, we have proposed a neuro-symbolic approach that leverages LLMs to generate auxiliary lemmas for inductive reasoning. Through a three-stage workflow, we synergistically integrate LLMs and constraint solvers, design prompt strategies to guide LLMs in generating high-quality candidate lemmas, and filter and validate them systematically. The experimental results on diverse benchmarks demonstrate the effectiveness of our approach.

In the future, extending the approach to handle more general proof tasks and investigating other prompting and inference-time techniques, or agent-based methods, would be promising.

Acknowledgments. This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320101. T. Chen is partially supported by overseas grants from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2023A04, KFKT2025A05).

References

1. Alhessi, Y., Einarsdóttir, S.H., Granberry, G., First, E., Johansson, M., Lerner, S., Smallbone, N.: Lemmanaid: Neuro-symbolic lemma conjecturing (2025), https://arxiv.org/abs/2504.04942
2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I. vol. 13243, pp. 415–442 (2022)
3. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
4. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA@ECOOP. pp. 1:1–1:10. ACM (2013). https://doi.org/10.1145/2489837.2489838
5. Cao, W., Wu, G., Xu, T., Yao, Y., Wei, H., Chen, T., Ma, X.: Clause2inv: A generate-combine-check framework for loop invariant inference. Proc. ACM Softw. Eng. **2**(ISSTA), 1009–1030 (2025). https://doi.org/10.1145/3728920, https://doi.org/10.1145/3728920
6. Chuharski, J., Collins, E.R., Meringolo, M.: Mining math conjectures from LLMs: A pruning approach (2024), https://arxiv.org/abs/2412.16177
7. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Proceedings of the 24th International Conference on Automated Deduction (CADE). vol. 7898, pp. 392–406. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_27
8. Cruanes, S.: Superposition with structural induction. In: Proceedings of the 11th International Symposium on Frontiers of Combining Systems (FroCoS). vol. 10483, pp. 172–188. Springer (2017). https://doi.org/10.1007/978-3-319-66167-4_10
9. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Solving horn clauses on inductive data types without induction. Theory Pract. Log. Program. **18**(3-4), 452–469 (2018). https://doi.org/10.1017/S1471068418000157, https://doi.org/10.1017/S1471068418000157
10. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Removing algebraic data types from constrained horn clauses using difference predicates. In: Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR). vol. 12166, pp. 83–102. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_6
11. Dixon, L., Fleuriot, J.D.: Higher order rippling in IsaPlanner. In: Slind, K., Bunker, A., Gopalakrishnan, G. (eds.) Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs). vol. 3223, pp. 83–98. Springer (2004). https://doi.org/10.1007/978-3-540-30142-4_7
12. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Proceedings of the 26th International Conference on Computer Aided Verification (CAV). vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
13. Echenim, M., Peltier, N.: Combining induction and saturation-based theorem proving. J. Autom. Reason. **64**(2), 253–294 (2020). https://doi.org/10.1007/S10817-019-09519-X

14. Gauthier, T., Urban, J.: Learning conjecturing from scratch (2025), https://arxiv.org/abs/2503.01389
15. Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al.: Deepseek-coder: When the large language model meets programming–the rise of code intelligence. arXiv preprint arXiv:2401.14196 (2024)
16. Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: Proceedings of the 13th International Conference on Intelligent Computer Mathematics (CICM). vol. 12236, pp. 123–137. Springer (2020). https://doi.org/10.1007/978-3-030-53518-6_8
17. Hajdú, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: Proceedings of the Conference on Formal Methods in Computer Aided Design (FMCAD). pp. 1–10. IEEE (2021). https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_34
18. Hajdú, M., Kovács, L., Rawson, M., Voronkov, A.: The vampire approach to induction (short paper). In: Proceedings of the Workshop on Practical Aspects of Automated Reasoning (IJCAR). vol. 3201 (2022)
19. Hamza, J., Voirol, N., Kuncak, V.: System FR: formalized foundations for the stainless verifier. Proc. ACM Program. Lang. **3**(OOPSLA), 166:1–166:30 (2019). https://doi.org/10.1145/3360592
20. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. In: Proceedings of the 28th International Conference on Automated Deduction (CADE). vol. 12699, pp. 361–377. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_21
21. Ireland, A.: Productive use of failure in inductive proof. J. Autom. Reason. **16**(1-2), 79–111 (1996). https://doi.org/10.1007/BF00244460, https://doi.org/10.1007/BF00244460
22. K., H.G.V., Shoham, S., Gurfinkel, A.: Solving constrained horn clauses modulo algebraic data types and recursive functions. Proceedings of the ACM on Programming Languages **6**(POPL), 1–29 (2022). https://doi.org/10.1145/3498722
23. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-aided reasoning: ACL2 case studies, vol. 4. Springer Science & Business Media (2013)
24. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016). https://doi.org/10.1007/S10703-016-0249-4
25. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Proceedings of the 25th International Conference on Computer Aided Verification (CAV). vol. 8044, pp. 1–35. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_1
26. Kurashige, C., Ji, R., Giridharan, A., Barbone, M., Noor, D., Itzhaky, S., Jhala, R., Polikarpova, N.: CCLemma: E-graph guided lemma discovery for inductive equational proofs. Proc. ACM Program. Lang. **8**(ICFP), 818–844 (2024)
27. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. Proc. ACM Program. Lang. **7**(OOPSLA1), 286–315 (2023). https://doi.org/10.1145/3586037
28. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Revised Selected Papers. vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20

29. Li, L., Sleem, L., Gentile, N., Nichil, G., State, R.: Exploring the impact of temperature on large language models: Hot or cold? CoRR **abs/2506.07295** (2025). https://doi.org/10.48550/ARXIV.2506.07295
30. Lv, K., Dong, Y., Han, R., Jia, F., Ma, F., Zhang, J.: Llm-guided quantified SMT solving over uninterpreted functions (2026), https://arxiv.org/abs/2601.04675
31. Ma, L., Liu, S., Li, Y., Xie, X., Bu, L.: Specgen: Automated generation of formal program specifications via large language models. In: Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 16–28 (2025). https://doi.org/10.1109/ICSE55347.2025.00129
32. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Proceedings of the 28th International Conference on Automated Deduction (CADE). vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
33. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Paulson, L.C.: Isabelle: A generic theorem prover. Springer (1994)
35. Peled, R., Kroening, D., Tautschnig, M., Vizel, Y.: Large lemma miners: Can LLMs do induction proofs for hardware? (2025), https://arxiv.org/abs/2511.02521
36. Pirzada, M.A.A., Reger, G., Bhayat, A., Cordeiro, L.C.: Llm-generated invariants for bounded model checking without loop unrolling. In: Filkov, V., Ray, B., Zhou, M. (eds.) Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024. pp. 1395–1407. ACM (2024). https://doi.org/10.1145/3691620.3695512, https://doi.org/10.1145/3691620.3695512
37. Reger, G., Suda, M., Voronkov, A.: Instantiation and pretending to be an SMT solver with vampire. In: Brain, M., Hadarean, L. (eds.) Proceedings of the 15th International Workshop on Satisfiability Modulo Theories affiliated with the International Conference on Computer-Aided Verification (CAV 2017), Heidelberg, Germany, July 22 - 23, 2017. CEUR Workshop Proceedings, vol. 1889, pp. 63–75. CEUR-WS.org (2017), https://ceur-ws.org/Vol-1889/paper6.pdf
38. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: Proceedings of the 27th International Conference on Automated Deduction (CADE). vol. 11716, pp. 477–494. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_28
39. Ren, Z.Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z.F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., Ruan, C.: Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition (2025), https://arxiv.org/abs/2504.21801
40. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). vol. 8931, pp. 80–98. Springer (2015). https://doi.org/10.1007/978-3-662-46081-8_5
41. Silva, Á., Mendes, A., Ferreira, J.F.: Leveraging large language models to boost Dafny's developers productivity. In: Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 138–142 (2024)

42. Singher, E., Itzhaky, S.: Theory exploration powered by deductive synthesis. In: Proceedings of the 33rd International Conference on Computer Aided Verification (CAV), Part II. vol. 12760, pp. 125–148. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_6
43. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). vol. 7214, pp. 407–421. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_28
44. Sun, Y., Ji, R., Fang, J., Jiang, X., Chen, M., Xiong, Y.: Proving functional program equivalence via directed lemma synthesis. In: Proceedings of the 26th International Symposium on Formal Methods (FM), Part I. vol. 14933, pp. 538–557. Springer (2024)
45. Sun, Z., Du, X., Song, F., Wang, S., Li, L.: When neural code completion models size up the situation: Attaining cheaper and faster completion through dynamic model inference. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE). pp. 1–12 (2024)
46. Sun, Z., Du, X., Song, F., Wang, S., Ni, M., Li, L., Lo, D.: Don't complete it! preventing unhelpful code completion for productive and sustainable neural code completion systems. ACM Transactions on Software Engineering and Methodology **34**(1), 1–22 (2025)
47. Varambally, S., Voice, T., Sun, Y., Chen, Z., Yu, R., Ye, K.: Hilbert: Recursively building formal proofs with informal reasoning (2025), https://arxiv.org/abs/2509.22819
48. Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M.D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxcé, H., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W.D., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., Li, J.: Kimina-prover preview: Towards large formal reasoning models with reinforcement learning (2025), https://arxiv.org/abs/2504.11354
49. Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S.C., Tian, C.: Enchanting program specification synthesis by large language models using static analysis and program verification. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 302–328. Springer (2024)
50. Wu, G., Cao, W., Yao, Y., Wei, H., Chen, T., Ma, X.: LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In: Filkov, V., Ray, B., Zhou, M. (eds.) Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024. pp. 406–417. ACM (2024). https://doi.org/10.1145/3691620.3695014, https://doi.org/10.1145/3691620.3695014
51. Yang, C., Li, X., Misu, M.R.H., Yao, J., Cui, W., Gong, Y., Hawblitzel, C., Lahiri, S.K., Lorch, J.R., Lu, S., Yang, F., Zhou, Z., Lu, S.: Autoverus: Automated proof generation for rust code. Proc. ACM Program. Lang. **9**(OOPSLA2), 3454–3482 (2025). https://doi.org/10.1145/3763174, https://doi.org/10.1145/3763174
52. Yang, G., Zhou, Y., Chen, X., Zhang, X., Zhuo, T.Y., Chen, T.: Chain-of-thought in neural code generation: From and for lightweight language models. IEEE Trans. Software Eng. **50**(9), 2437–2457 (2024)
53. Yang, W., Fedyukovich, G., Gupta, A.: Lemma synthesis for automating induction over algebraic data types. In: Schiex, T., de Givry, S. (eds.) Proceedings of the

25th International Conference on Principles and Practice of Constraint Programming (CP). vol. 11802, pp. 600–617. Springer (2019). https://doi.org/10.1007/978-3-030-30048-7_35