Primal Grammars Driven Automated Induction

Adel Bouhoula[1] and **Miki Hermann**[2]
[1]Arabian Gulf University, Bahrain
[2]LIX, CNRS, École Polytechnique, Institut Polytechnique de Paris, France
a.bouhoula@agu.edu.bh, hermann@lix.polytechnique.fr

Abstract

Automated induction is a powerful method for the validation of critical systems. However, the inductive proof process faces major challenges: it is undecidable and diverges even with small examples.

Previous methods have proposed ad-hoc heuristics to speculate on additional lemmas that hopefully stop the divergence. Although these methods have succeeded in proving interesting theorems, they have significant limitations: in particular, they often fail to find appropriate lemmas, and the lemmas they provide may not be valid.

We present a new method that allows us to perform inductive proofs in conditional theories. This method automatically detects divergence in proof traces and derives primal grammars as well as new lemmas that schematize the divergent sequence. This new construction allows us to break the divergence and complete the proof.

Our method is presented as a set of inference rules whose soundness and refutational completeness have been formally proved. Unlike previous methods, our method is fully automated and has no risk of over-generalization. Moreover, our technique for capturing and schematizing divergence represents the most general decidable schematization, with respect to description power, among all known schematizations.

Our method has been implemented in C++ and successfully proved over fifty complex examples that fail with well-known theorem provers (e.g., ACL2, Isabelle, PVS, SPIKE) and related methods for handling divergence in proofs by induction.

Our method represents a significant contribution to the field of automated reasoning as it can be integrated with existing automated and interactive inductive proof systems to enhance their performance. Moreover, it has the potential to substantially reduce the time needed for the verification of critical systems.

1 Introduction

Proof by induction is more and more frequently adopted for certifying critical hardware and software. Its power is expressed by the successes of the Boyer-Moore theorem prover Nqthm [Boyer and Moore, 1979], that has been considered for many years as the only significant automated theorem proving system for induction. Other inductive theorem provers have also been developed like SPIKE [Bouhoula and Rusinowitch, 1993], RRL [Kapur and Zhang, 1995], PVS [Rushby *et al.*, 1998], ACL2 [Kaufmann *et al.*, 2000] (successor version of Nqthm), LEAN [de Moura and Ullrich, 2021], and Vampire [Hozzová *et al.*, 2021]. Unfortunately, very often they present an undesirable phenomenon called *divergence*, where new subgoals are generated over and over again, without finally terminating with the required goal, which is a proof or a disproof of the initial conjectures. This phenomenon is present even in the case of very small examples.

Previous methods have proposed ad-hoc heuristics to speculate on additional lemmas that hopefully stop the divergence. Although these methods have succeeded in proving several interesting conjectures, they have significant limitations: the provided lemmas may not be valid in the initial model of the axioms, and they often fail to find appropriate lemmas. In this case, the user will manually try to find the necessary lemmas. It is a very tedious process, and results are not guaranteed.

Our approach is new in several aspects. The proof procedure does not require to be stopped but proceeds automatically by detecting the divergence of proof traces and deriving a *primal grammar* that schematizes the divergent sequences. It suggests some new lemmas that subsume all conjectures of the divergent sequence and allow the proof procedure to automatically discard them, continuing the proof with the new lemmas. Our method is presented as a set of inference rules whose soundness and refutational completeness has been formally proved. Refutational completeness is particularly useful for debugging programs and detecting flaws in security protocols.

Since divergence is undecidable in general, we do not pretend that our procedure produces a result in every case. However, if the divergent sequence of clauses, produced during an inductive proof, follows a primitive recursive pattern, our approach always produces a correct and complete schematization by a primal grammar, which allows to stop the divergent process and successfully terminate the proof. Integrating primal grammars and the new lemmas into the proof procedure preserves refutational completeness and avoids the risk of over-generalization. Our method has been implemented in C++ and successfully proved over fifty complex examples in a completely automatic way, unlike related provers, which require user interaction or ad hoc heuristics or fail, since it is not always possible to speculate on additional lemmas to complete a proof.

The main advantages of our method compared to related work are: (i) it is completely automatic, i.e., no interaction with the user, (ii) it deals with conditional axioms, (iii) it preserves refutational completeness, (iv) there is no risk of over-generalization,

Our method represents a major development in the field of automated reasoning as it can be integrated with existing automated and interactive inductive proof systems to enhance their performance. Moreover, computer experiments show that our method has the potential to significantly reduce the time needed for the verification of critical systems.

The paper is organized as follows. Section 2 presents related work for both solving the divergent issues of proofs by induction and schematization of divergence, together with our motivation. Section 3 introduces the necessary preliminaries and notation. In Section 4 we define our method of primal grammars for capturing divergence, together with the algorithm generating them from a divergent sequence of clauses. Section 5 explains the construction of a primal grammar from a divergent sequence. Section 6 introduces inductive positions and test sets. Section 7 presents our inference system to perform inductive proofs with the help of primal grammars. Section 8 summarizes the results of our computer experiments and shows the effectiveness of our new method through two illustrative examples. More examples can be found at the github page https://github.com/BouhoulaHermann/IJCAI-2024.

2 Related Work

Several methods have been proposed to avoid or stop the divergence of rewriting-based inductive provers in unconditional theories. Most of them consist of stopping the prover in his divergent behavior and either manually or using ad-hoc heuristics to speculate on additional lemmas or to discover generalizations, which hopefully stop this unwanted phenomenon: the primary term heuristic [Aubin, 1976], the rippling heuristic [Bundy *et al.*, 1993], the difference matching procedure [Basin and Walsh, 1992; Basin and Walsh, 1993], the divergence critic [Walsh, 1996], the use of proof planning [Ireland and Bundy, 1996], the speculate heuristic [Kapur and Subramaniam, 1996], the generalization of conjectures [Urso and Kounalis, 2004], and the induction with generalization [Hajdú *et al.*, 2020].

Although these methods have succeeded to prove several interesting theorems, they have important limitations: First, they may require high user guidance. Second, there are no guarantees against over-generalization (except in [Urso and Kounalis, 2004] under some conditions) and provided lemmas may not be valid in the initial model of the axioms. Third, attempting to prove the lemmas themselves can cause fresh divergence.

Several researchers in the 1990s noted that the infinite behavior during theorem proving, called *divergence*, follows a primitive recursive pattern. To cope with this phenomenon, they proposed different *schematizations*. Most notable among them and in their order of increasing schematization power are those of [Chen *et al.*, 1990], [Salzer, 1992], [Comon, 1995], and [Hermann and Galbavý, 1997].

However, up till now nobody proposed to synthesize *primal grammars* from divergent proofs. Subsequently, there is no previous work on maintaining refutational completeness in the presence of schematizations.

To guarantee that our method can handle a very large class of specifications and conjectures, we have developed a new version of primal grammars based on Presburger arithmetic. These grammars represent the most general decidable schematization, with respect to description power, among all known schematizations.

Certainly, as the divergence of equational proofs is undecidable in general, not all divergent sequences can be schematized by primal grammars. We can only consider those presenting a primitive recursive behavior, which allows us to cover a large class of problems. It is important to note that the Ackermann function for example fails to be captured by primal grammars since it is not compatible with primitive recursion.

In this paper, we propose a new method for proof by induction in conditional theories. Our method automatically detects the divergence of proof traces and derives a primal grammar, as well as new lemmas that subsume all conjectures of the divergent sequence. This allows the proof procedure to automatically discard them and continue the proof with the new lemmas.

Our method has been implemented and the numerous computer experiments show that our method is very promising and can significantly contribute to shortening the time needed to validate critical systems.

3 Preliminaries

We introduce the main notation used later and refer to [Dershowitz and Jouannaud, 1991; Bouhoula, 1997; Baader and Nipkow, 1998; Bouhoula, 2009] for the details and missing definitions. We always assume that we are working within a quantifier-free first-order language with equality.

A many-sorted signature Σ is a pair (S, F) where S is a set of sorts and F is a finite set of function symbols, partitioned into two disjoint subsets: the first one C, contains the constructor symbols and the second D, is the set of defined symbols. Let X be a countably infinite set of variables and let $T(F, X)$ be the set of well-sorted *F-terms*. A *ground* term has no variables. $T(F)$ denotes the set of ground terms. A *position* a of a term $t \in T(F, X)$ is a string of natural numbers which is an index of a subterm in a tree representing t. $\mathrm{Pos}(t)$ denotes the set of all positions of a term t. $\mathrm{FPos}(t)$ denotes the subset of $\mathrm{Pos}(t)$ containing only the functional positions in t. Two positions a and b of a term t are *parallel* if a is not a prefix of b, nor b is a prefix of a. A set of positions A is called *parallel* if any two positions $a, b \in A$ are parallel. A *subterm* of t at position $a \in \mathrm{Pos}(t)$ is denoted by $t|_a$.

An *equation* is a pair of the form $t_1 = t_2$, where t_1 and t_2 are terms of the same sort. A conditional equation is either an equation or an expression of the form $e_1 \wedge \cdots \wedge e_n \Rightarrow e$, where $e, e_1, \ldots, e_n$ are equations, with $e_1, \ldots, e_n$ being the conditions and e is the conclusion. A clause is an expression of the form $\neg e_1 \vee \cdots \vee \neg e_n \vee e'_1 \vee \cdots \vee e'_m$. *Axioms* are built from conditional equations. *Goals* to be proved are clauses.

A term t is an *instance* of t' if there is a substitution σ such that $t = t'\sigma$. A *variable renaming* is an injective substitution $\sigma\colon X \to X$, such that its domain and range are disjoint for a finite subset of variables $X' \subset X$, for which $x\sigma \neq x$ holds. Two terms s and t are *equivalent* modulo variable renaming if s is an instance of t and t is an instance of s.

We denote by $\mathrm{Vars}(t)$ the variables of a term t. A *rewrite rule* $l \to r$ is an oriented equality such that $l \in T(F, X) \setminus X$ and $\mathrm{Vars}(r) \subseteq \mathrm{Vars}(l)$, where $\mathrm{Vars}(t)$ is the set of variables of t. In addition, we require the relation $l \succ r$ to hold where $\succ$ is a well-founded ordering on terms. A *term rewriting system* is a set of (conditional) rewrite rules. Given a term rewriting system R, a reduction relation $\to_R$ on terms is defined as $s \to_R t$ if there is a (conditional) rule $c \Rightarrow l \to r$ and a substitution σ, such that $l \succ c$, $l\sigma$ is a subterm of s and for each equation in the condition c of the form $a = b$, the instances $a\sigma$ and $b\sigma$ reduce to the same normal form, Then t is obtained by replacing the subterm $l\sigma$ by $r\sigma$ in s. We denote by $t{\downarrow_R}$ the *normal form* of the term t obtained by reduction using a confluent and terminating rewrite system R.

A term t is *R-irreducible* if there is no term s such that $t \to_R s$. A substitution σ is R-irreducible if $x\sigma$ is R-irreducible for any variable x of its domain. We say that two terms s and t are *joinable* if $s \to_R^\star v$ and $t \to_R^\star v$ hold for some term v. A formula φ is a deductive theorem of E if it is valid in any model of E, denoted by $E \models \varphi$. The rewrite relation $\to_R$ is said to be *ground convergent* if the terms u and v are joinable whenever $u, v \in T(F)$ and $R \models u = v$.

Given a set of axioms E and a clause c called a *conjecture*, we want to prove that c is an *inductive consequence* of E, i.e., any ground instance of c is a consequence of E. In order to establish consequence relations, we consider initial models. We write $E \models_{ind} c$, if c is valid in the *initial* model of E. Any ground instance $c\sigma$ such that $E \not\models c\sigma$ is a *counterexample*. A clause c is not valid in the initial model of E if there is a ground instance of c which is a counterexample. The deduction relation $\models_{ind}$ extends to sets of equalities as follows: $E \models_{ind} \Phi$ holds if $E \models_{ind} e$ for every $e \in \Phi$.

Let $T(C)$ be the set of ground terms constructed from the signature C. An operator $f \in D$ is *sufficiently complete* if for all $t_1, \ldots, t_n \in T(C)$, there exists $t \in T(C)$ such that $f(t_1, \ldots, t_n) \to_R^\star t$. If each $f \in D$ is sufficiently complete then we say that R is *sufficiently complete*. Let $f \in D$ be a sufficiently complete defined function. If for all rewriting rules $c_i \Rightarrow f(t_1, \ldots, t_n) \to r_i$ whose left-hand sides are identical up to a variable renaming σ_i, we have $c_1\sigma_1 \vee \cdots \vee c_n\sigma_n$ valid in the initial model of R, then f is *strongly complete* with respect to R. We say that R is *strongly complete* if any function symbol in D is strongly complete with respect to R.

4 Capturing Divergence

Divergence in equational proofs is undecidable in general, since we can simulate the work of a Turing machine by means of superposition and reduction during a proof derivation. This fact implies that we can search only for sufficient conditions to detect and capture divergence. An appropriate tool for capturing divergence is the formalism of *primal grammars* [Hermann and Galbavý, 1997]. Primal grammars are based on primitive recursive functions adapted for term rewriting systems. They represent the most powerful schematization of divergent systems in terms of inclusion. However, if a proof derivation presents a fast-growing behavior beyond the class of primitive recursive functions, like for instance the Ackermann function, there exists no reasonable general purpose mean to cope with such type of divergence, except the design of an ad-hoc schematization.

For the needs of primal grammars, the signature F consists of *constructors* K, *defined symbols* D, where K and D are disjoint, and the special symbols $+$ (addition), plus the constants 0 and 1. The defined symbols are surmounted by a hat to distinguish them from the constructors. We also need the set of *counter variables* C, disjoint from ordinary variables X. Note that constructors for primal terms are different from constructors in inductive proofs.

The arguments of the defined symbols are divided into two parts by a semicolon. Those before the semicolon are called *counters*. Each defined symbol $\hat{f}$ has a *counter arity* $\mathrm{car}(\hat{f})$. The manipulation of counters is based on the *counter expressions* $\mathbb{N}(C)$. They are the smallest algebra containing 0, the counter variables C, and closed under the application of addition by 1. Since this algebra is isomorphic to linear expressions over natural numbers, we use the arithmetic expressions $c + 1$ and $c + k$ for the addition by 1 and the addition by k.

The schematization level is represented by the algebra of *primal terms* $P = (K, D; C, X)$, which is the smallest set such that:

(1) $X \subseteq P$,
(2) if $c_1, \ldots, c_k \in \mathbb{N}(C)$, $t_1, \ldots, t_n \in P$, and $\hat{f} \in D$ with $\mathrm{car}(\hat{f}) = k$ and $\mathrm{ar}(\hat{f}) = n$, then $\hat{f}(c_1, \ldots, c_k; t_1, \ldots, t_n) \in P$,
(3) if $t_1, \ldots, t_n \in P$ and $f \in K$ with $\mathrm{ar}(f) = n$ then $f(t_1, \ldots, t_n) \in P$.

The *approximation* of $\hat{f}(\boldsymbol{c}; \boldsymbol{t})$ with respect to a precedence $\succ$ on D is the set $\mathrm{Apx}(\hat{f}(\boldsymbol{c}; \boldsymbol{t})) = \{\hat{g}(\boldsymbol{z}; \boldsymbol{u}) \mid \hat{f} \succ \hat{g}\}$, where $\boldsymbol{z}$ is a subsequence of $\boldsymbol{c}$ and $\boldsymbol{u}$ is a subsequence of $\boldsymbol{t}$.

The set of positions occupied by defined symbols in a term t is denoted by $\mathrm{DPos}(t)$. A *wrap* of a primal term t is the context $\mathrm{Wrp}(t) = t[\cdot]_{\mathrm{DPos}(t)}$ where each subterm at every position $\mathrm{DPos}(t)$ of a defined symbol is replaced by a *hole*, a special constant absent from any set of symbols F. A depth d *iteration* (or d-iteration) of a wrap $w = w[\cdot]_A$, denoted by $w\langle A\rangle^d$ for $d \in \mathbb{N}$, is defined as $w\langle A\rangle^1 = w$ and $w\langle A\rangle^{i+1} = w[w\langle A\rangle^i]_A$. A d-iteration of a wrap $w = w[\cdot]_A$ modulo variable renaming, denoted by $w^{\sim d}\langle A\rangle^d$, is recursively defined by $w^{\sim 1}\langle A\rangle^1 = w$ and $w^{\sim i+1}\langle A\rangle^{i+1} = w_{i+1}[w^{\sim i}\langle A\rangle^i]_A$, where all $w_i[\cdot]_A$, $i \in \mathbb{N}$, are equivalent modulo variable renamings with disjoint ranges.

Primal terms are constructed to represent infinite sets of terms. Their semantics is defined by *Presburger systems.*

Definition 1. Let $\succ$ be the precedence on the defined symbols D. With every defined symbol $\hat{f}$ we associate two primal terms r_1 and r_2. The **Presburger rewrite system** $\mathcal{R}$ contains for each $\hat{f} \in D$ the following pair of rewrite rules:

- the *basic rule* $\hat{f}(0; \boldsymbol{x}) \to r_1$ or $\hat{f}(0, c; \boldsymbol{x}) \to r_1$
- and the *inductive rule*, having one of the following forms:
  $\hat{f}(n+1; \boldsymbol{x}) \to r_2[\hat{f}(n; \boldsymbol{x})]_A$ or
  $\hat{f}(n+1, c; \boldsymbol{x}) \to r_2[\hat{f}(n, c; \boldsymbol{x})]_A$ or
  $\hat{f}(n+1, c; \boldsymbol{x}) \to r_2[\hat{f}(n, c+1; \boldsymbol{x})]_A$.

where

- c is a counter variable, and
- $\boldsymbol{x}$ is a vector of ordinary variables,
- A is a subset of parallel non-root positions in r_2,
- both terms r_1 and r_2 at every position from $\mathrm{DPos}(r_1)$ and $\mathrm{DPos}(r_2)$ belong to $\mathrm{Apx}(\hat{f}(n, \boldsymbol{c}; \boldsymbol{x}))$,
- the root symbol of r_2 is a constructor,
- each ordinary variable in $\boldsymbol{x}$ occurs in $\mathrm{Wrp}(r_1) \cup \mathrm{Wrp}(r_2)$.

A Presburger system is designed as a primitive recursive function over the set of terms $T(F, X)$. Each Presburger system is confluent and terminating [Hermann and Galbavý, 1997].

An *enumerator* for a primal term t is a ground substitution $\xi \colon C \to \mathbb{N}$ that instantiates all counter variables of t by natural numbers. The *enumeration* $\Xi(t)$ is the set of all possible enumerators of t. A primal term represents the folded form of all terms in the schematized set S and an enumerator selects one element from S. A Presburger system $\mathcal{R}$ provides us with the machinery for computing the elements of a schematized set from a primal term t by instantiation $t\xi$ by a enumerator $\xi \in \Xi(t)$, followed by reduction to normal form $t\xi{\downarrow_{\mathcal{R}}}$.

Definition 2. A **primal grammar** is a quadruple $G = (K, D, \mathcal{R}, \bar{t})$, where K are the constructors, D are the defined symbols, $\mathcal{R}$ is a Presburger rewrite system, and $\bar{t}$ is a *lemma*. The language generated by a primal grammar is the set $L(G) = \{\bar{t}\xi{\downarrow_{\mathcal{R}}} \mid \xi \in \Xi(\bar{t})\}$. We say that the primal grammar G *schematizes* the set $L(G)$.

The lemma $\bar{t}$ in the aforementioned definition can also be an equation or a clause, as it is the case in the scope of theorem proving. We put a bar on top of the primal term to distinguish it as the starting point of the primal grammar.

We need to define the sets of terms to which primal grammars correspond.

Definition 3 (Primitive Recursive Set of Terms)**.** A countably infinite set of terms T is called **0-level primitive recursive** if it can be arranged in an infinite sequence $t_0, t_1, t_2, \ldots, t_n, \ldots$ and there exists a set of parallel positions A, such that

- r_1 is the first term t_0 of the set T,
- the equality $t_i = t_{i+1}|a$ holds for every $i \in \mathbb{N}$ and each position $a \in A$,
- there exists a wrap $r_2[\cdot]_A$ such that $t_i\langle A\rangle^i = r_2^{\sim i}\langle A\rangle^i$.

A countably infinite set of terms T is called **ℓ-level primitive recursive** if it can be arranged in an infinite sequence t_0, t_1, t_2, …, t_n, … and there exists a set of parallel positions A, such that

- r_1 is the first term t_0 of the set T,
- the equality $t_i = t_{i+1}|a$ holds for every $i \in \mathbb{N}$ and each position $a \in A$,
- there exists a superset of parallel positions $B \supseteq A$,
- there exists a wrap $r_2[\cdot]_B$ such that $t_i\langle B\rangle^i = r_2^{\sim i}\langle B\rangle^i$,
- for each position $b \in B \setminus A$ there exists a k-level primitive recursive set of terms T^b, where $k < \ell$,
- $t_i|_b = t_i^b$ holds for every $i \in \mathbb{N}$ and each position $b \in B \setminus A$, where $t_i^b \in T^b$ is the ith term in the sequence of T^b.

A **concatenation** of an ℓ-level and a k-level primitive recursive sets $T^\ell = \{t_0^\ell, t_1^\ell, \ldots\}$ and $T^k = \{t_0^k, t_1^k, \ldots\}$ with $k < \ell$ is a primitive recursive set $T' = \{t'_0, t'_1, \ldots\}$ if there exists two sets of parallel positions A and B, such that $t'_i = t_i^\ell\langle A\rangle^i[t_i^k]_B$ for each $i \in \mathbb{N}$.

A **combination** of a finite number of primitive recursive sets T_0, T_1, …, T_n is a primitive recursive set $T = \{t_0, t_1, \ldots\}$ if there exists mutually parallel sets of parallel positions A_i and a wrap $w[\cdot]_A$ such that $t_i = w[t_i^0]_{A_0} \cdots [t_i^n]_{A_n}$ where t_i^j is the ith term in the sequence T_j for $j = 0, \ldots, n$.

A countably infinite set of terms T is **primitive recursive** if it is ℓ-level primitive recursive for some $\ell \in \mathbb{N}$, a concatenation, or a combination of primitive recursive sets.

Remark. Experimental observations show that each divergent sequence of clauses in an inductive proof is a primitive recursive set. The requirement that consecutive wraps must be equivalent *modulo variable renaming* takes care of *rolling variables* which may occur by instantiation of induction variables by the terms from the test sets (see Section 6). In the concatenation of two primitive recursive sets T^ℓ and T^k a term $t_i \in T^\ell$ is paired with a term $t'_i \in T^k$ at the same index i.

Note also that all (ordinary) variables in each primitive recursive set T are always considered Skolemized, i.e., they cannot be instantiated.

Example 4. The following finite prefix of a primitive recursive set of terms T has been produced during an inductive proof of the conjecture $len(app(x, x)) = len(x) + len(x)$ over a specification of a list with the operations zero 0, successor s, addition $+$, empty list nil, concatenation $cons$, and application app. We obtained the following equations:

$$\begin{aligned}
&len(app(y, cons(x_1, y))) = s(len(y)) + len(y),\\
&len(app(y, cons(x_2, cons(x_1, y)))) =\\
&\qquad\qquad s(s(len(y))) + len(y),\\
&len(app(y, cons(x_3, cons(x_2, cons(x_1, y))))) =\\
&\qquad\qquad s(s(s(len(y)))) + len(y).
\end{aligned}$$

There is first the 0-level primitive recursive set $T_0 = \{s(len(y)), s(s(len(y))), s(s(s(len(y)))), \ldots\}$ with the set of positions[1] $A = \{0\}$ and the wrap $s(\cdot)$. There is another 0-level primitive recursive set

$$\begin{aligned}
T_1 = \{&cons(x_1, y), \quad cons(x_2, cons(x_1, y)),\\
&cons(x_3, cons(x_2, cons(x_1, y))), \ldots\}
\end{aligned}$$

with the set of positions $A = \{1\}$ and the wrap $cons(x, \cdot)$ with variable renamings $\sigma_i = [x \leftarrow x_i]$. They are then combined into the resulting primitive recursive set using the wrap

[1]All indices begin with 0.

$len(app(y,\cdot)) = s(\cdot) + len(y)$. The variable x taking the indices 1, 2, 3, etc, is a so-called *rolling variable*.

The following finite prefix of a primitive recursive set of terms T has been produced during an inductive proof of the associativity conjecture $x + (x + x) = (x + x) + x$ over a specification of natural numbers with the operations of zero 0, successor s, and addition $+$. We obtained the equations

$$x + s(x + s(x)) = (x + s(x)) + s(x),$$
$$x + s^2(x + s^2(x)) = (x + s^2(x)) + s^2(x),$$
$$x + s^3(x + s^3(x)) = (x + s^3(x)) + s^3(x).$$

There is first the 0-level primitive recursive set $T_0 = \{s(x), s^2(x), s^3(x), \ldots\}$, followed by a 0-level primitive recursive set $T_1 = \{s(x+y),\ s^2(x+y),\ s^3(x+y)\}$, both with positions $A = \{0\}$ and wrap $s(\cdot)$. They are concatenated into a 1-level primitive recursive set $T_2 = \{s(x + s(x)), s^2(x + s^2(x)), s^3(x + s^3(x)), \ldots\}$ using the position $B = \{0.1\}$. Finally the sets T_0 and T_2 are combined using the wrap $x + \cdot = (x + \cdot) + \cdot$ into the resulting primitive recursive set, where the first hole is replaced by T_2 and the following two holes each by T_0.

Proposition 5. *For each primitive recursive set of terms T there exists a primal grammar G_T, such that $L(G_T) = T$.*

Hint. Induction on levels $\ell \in \mathbb{N}$. Each level is assigned a different defined symbol $\hat{f}$. Descent to lower level is taken care of in the wrap $r_2[\cdot]_A$, concatenation is taken care of in r_1, combination is taken care of in the lemma $\bar{t}$. □

Proposition 6. *For each primal grammar G there exists a primitive recursive set of terms T such that $L(G) = T$.*

Hint. Direct construction. The levels are determined by the ordering $\succ$ on defined symbols D. □

5 Constructing a Primal Grammar

Hermann and Galbavý defined primal grammars in [Hermann and Galbavý, 1997] and developed a unification algorithm for them, but they did not supply any procedure allowing to derive a suitable correct and complete primal grammar from an existing divergent sequence. In this section, we will formally develop the PRIMGRAM procedure performing exactly this necessary derivation.

The first step in taming divergence is the design of a primal grammar $G_T = (K_T, D_T, \mathcal{R}_T, \bar{t}_T)$ schematizing an infinite divergent sequence $T = \{t_1, t_2, \ldots\}$, such that $L(G_T) = T$. Let $T_p = \{t_1, t_2, \ldots, t_p\} \subset T$ be a finite prefix of p initial terms of the divergent sequence T. To determine the maximal common wrap of a finite set of terms, we use the machinery of *difference matching* [Basin and Walsh, 1992] resp. *difference unification* [Basin and Walsh, 1993].

The PRIMGRAM function for deriving a primal grammar for a divergent sequence is presented in Algorithm 1. The details of the PRIMGRAM function are described as follows:

(1) Determine the maximal common wrap $w[\cdot]_Q$ of the finite prefix T_p of the divergent sequence T by means of difference matching. This wrap consists of the term context common to all terms from T_p. Note that this wrap can have several holes on parallel positions Q.

(2) For each parallel position $q \in Q$, determine the corresponding divergent sequence of subterms $T_p^q = \{t_1^q, t_2^q, \ldots, t_p^q\}$. For each $i \in \{1, \ldots, p\}$, t_i^q is the subterm of t_i at position q. Formally, $t_i^q = t_i|_q$ for all $t_i \in T_p$.

(3) Using difference matching again, determine the maximal wrap $r_2[\cdot]_A$ of $T_p^q \smallsetminus \{t_1^q\}$ with holes at parallel positions A.

(4) For each pair t_i^q, t_{i+1}^q of consecutive terms from T_p^q, test if the subterm $t_{i+1}^q|_a$ at each position $a \in A$ is equal to t_i^q.

(5) If *not*, the finite prefix T_p^q of subterms T^q from T at position q hides an encapsulated divergence. Recursively call this procedure on it, solve the encapsulated divergence by producing a primal grammar $G' = (K', D', \mathcal{R}', \bar{t}')$ for T^q, and replace this encapsulated divergent sequence in all terms from T_p^q by the lemma $\bar{t}'$. If the recursive call returns a *failure*, return it.

(6) Check if the subterms $t_2^q|_a$ at all positions $a \in A$ are equal. If *not*, the maximal wrap $r_2[\cdot]_A$ is wrong and return *failure*. If *yes*, put $r_1 \leftarrow t_1^q$.

(7) Create the lemma $\bar{t}_T$ for primal grammar G_T from the maximal common wrap $w[\cdot]_Q$ by replacing the holes by corresponding primal term $\hat{f}_q(\boldsymbol{c}; \boldsymbol{x})$ for each position $q \in Q$.

(8) If there were encapsulated divergences encountered, which have now been replaced by the lemma $\bar{t}_e$ of their corresponding primal grammars G_e, incorporate the counters of $\bar{t}_e$ into actually created lemma $\bar{t}_{T^q}$. Integrate the dependent primal grammars G_e into the created primal grammar G_{T^q}. Perform it for all parallel positions $q \in Q$.

(9) By means of primal term matching of $\bar{t}_T$ against the prefix T_p, determine dependency of counters.

Definition 7. A primal grammar $G = (K, D, \mathcal{R}, \bar{t})$ is a **complete** schematization for a primitive recursive set of terms T if for each term $t \in T$ there exists an enumerator $\xi \in \Xi(\bar{t})$, such that $t = \bar{t}\xi{\downarrow_{\mathcal{R}}}$. A primal grammar $G = (K, D, \mathcal{R}, \bar{t})$ is a **correct** schematization for a primitive recursive set of terms T if for each enumerator $\xi \in \Xi(\bar{t})$ there exists a term $t \in T$, such that $t = \bar{t}\xi{\downarrow_{\mathcal{R}}}$.

The PRIMGRAM function for deriving a primal grammar for a divergent sequence is presented in Algorithm 1.

Theorem 8 (Correctness and Completeness)**.** *Let T be a primitive recursive set of terms. For each sufficiently long finite prefix $T_p \subset T$,* PRIMGRAM *generates a primal grammar G_T which is a correct and complete schematization of T.*

Theorem 8 presents the statement that the PRIMGRAM function produces a primal grammar G which neither overgeneralizes a divergent sequence T, nor omits a term from T, i.e., that $T = L(G)$ holds, provided that T is a primitive recursive set. The parameter p — sufficient length of the prefix T_p — is discussed further in the section on test sets.

The following example illustrates Theorem 8.

Example 9. Suppose that PRIMGRAM receives the divergent sequence of clauses $T = \{a^2(x) = true, a^4(x) = true, a^6(x) = true, \ldots\}$ which generalizes to $a^{2k+2}(x) = true$. We will show that PRIMGRAM neither over-generalizes the sequence T, nor omits a term from T.

Algorithm 1 Production of a primal grammar from a finite prefix of an infinite divergent sequence.

```
Input: Finite prefix $T_p = \{t_0, t_1, \ldots, t_{p-1}\}$ of an infinite divergent sequence $T$.
Output: Primal grammar $G_T = (K_T, D_T, \mathcal{R}_T, \bar{t}_T)$ or failure.
1:  function PRIMGRAM($T_p$)
2:      $w[\cdot]_Q \leftarrow maximal\_common\_wrap(T_p)$
3:      for all $q \in Q$ do
4:          $T_p^q \leftarrow \emptyset$
5:          for $i = 0, \ldots, p-1$ do $T_p^q \leftarrow T_p^q \cup \{t_i|_q\}$
6:          $r_2[\cdot]_A \leftarrow maximal\_common\_wrap(T_p^q \setminus \{t_0^q\})$
7:          for $i = 0, \ldots, p-2$ do
8:              for all $a \in A$ do
9:                  if $t_{i+1}^q|_a \neq t_i^q$ then
10:                     $(K', D', \mathcal{R}', \bar{t}') \leftarrow$ PRIMGRAM($T_p^q$)
11:                     if PRIMGRAM($T_p^q$) = failure then
12:                         return failure
13:                     for $j = 0, \ldots, p-1$ do $t_j^q|_a \leftarrow \bar{t}'$
14:                     break
15:         for all $a_1, a_2 \in A$ do
16:             if $t_2^q|_{a_1} \neq t_2^q|_{a_2}$ then return failure
17:         $r_1 \leftarrow t_0^q$
18:         $\bar{t}_T|_q \leftarrow w[\hat{f}_q(\boldsymbol{c}; \boldsymbol{x})]_q$
19:         $lift\_counters(\bar{t}_T|_q)$
20:     $match\_counters(\bar{t}_T, T_p)$
21:     return $(K_T, D_T, \mathcal{R}_T, \bar{t}_T)$
```

PRIMGRAM produces the Presburger system

$$\hat{f}_0(0;x) \to x, \qquad \hat{f}_0(n+1;x) \to a^2(\hat{f}_0(n;x))$$

and the lemma $c = (a^2(\hat{f}_0(n;x)) = true)$. For every enumerator ξ, instantiating the counter variable n in the lemma c, there exists a corresponding clause in T. This means that the lemma $a^2(\hat{f}_0(n;x)) = true$ subsumes the sequence T.

Suppose that PRIMGRAM generated the Presburger system

$$\hat{f}_0(0;x) \to x, \qquad \hat{f}_0(n+1;x) \to a(\hat{f}_0(n;x))$$

and the lemma $c = (a(\hat{f}_0(n;x)) = true)$ instead of the correct one. All clauses in T can be schematized by this primal grammar, but there exists, for instance, the enumerator $\xi = \{n \leftarrow 0\}$ for which the instance $c\{n \leftarrow 0\}$ reduces to the clause $a(x) = true$, which does not belong to T. A similar situation will happen with Presburger system whose inductive rule has the form $\hat{f}_0(n+1;x) \to a^{2k+1}(\hat{f}_0(n;x))$ for any k.

Suppose now that PRIMGRAM generated the Presburger system

$$\hat{f}_0(0;x) \to x, \qquad \hat{f}_0(n+1;x) \to a^4(\hat{f}_0(n;x))$$

instead of the correct one. We cannot produce for example the clause $a^4(x) = true$ which belongs to T. A similar situation will happen with any Presburger system whose inductive rule has the form

$$\hat{f}_0(n+1;x) \to a^{2k}(\hat{f}_0(n;x)) \qquad \text{for } k > 1.$$

We conclude that PRIMGRAM exactly captures all clauses in T. In this example, we assumed that T is infinite. However, in reality, PRIMGRAM takes only a finite prefix of T to generate the corresponding primal grammar. Since the sequence T contains a primitive recursive set below a maximal common wrap of all clauses, in this particular case of level 0, already a sufficiently long finite prefix T_p determines the required properties of the whole sequence T. According to Definition 3, we need to determine the set of parallel positions A, the wrap $r_2[\cdot]_A$ which iterates through all positions A, and the term t_0.

The maximal common wrap $w[\cdot]_Q$ of T is $a^2(\cdot) = true$ and Q contains a single position $q = 0.0.0$. This wrap will be used to construct the lemma. The terms T^q below the wrap $w[\cdot]_Q$ are $\{x, a^2(x), a^4(x), \ldots\}$. We need to show that T_q constitutes a primitive recursive set. The maximal common wrap $r_2[\cdot]_A$ of T^q is $a^2(\cdot)$ with $A = \{0.0\}$. We have $a^2(x)|_{0.0} = x$, $a^4(x)|_{0.0} = a^2(x)$, etc, and also the equality $a^{2i}(x)\langle 0.0\rangle^i = (a^2(x)^{\sim i}\langle 0.0\rangle^i$ for each i. The term r_1 is equal to t_0^q which is x. Therefore T^q is a 0-level primitive recursive set. From Definition 3 it follows that the set of parallel positions A, the wrap $r_2[\cdot]_A$, and the term r_1 can be determined from a finite prefix of the primitive recursive set T.

This example shows that PRIMGRAM generates exactly the equations of the form $a^{2k+2}(x) = true$ from a finite prefix of T.

The following theorem shows that PRIMGRAM preserves refutational completeness.

Theorem 10. *Let c be a lemma in the primal grammar $G = (K, D, \mathcal{R}, c)$ generated by* PRIMGRAM(T) *from a primitive recursive set T and let R be a ground convergent rewrite system. Then $R \models c$ implies $R \models T$.*

Hint. Since c subsumes the sequence T, $R \models_{ind} c$ implies $R \models_{ind} T$. This means that a counterexample for a clause in T is also a counterexample for c, and therefore we conclude that PRIMGRAM maintains the refutational completeness property. □

6 Inductive Positions, Induction Variables, and Test Sets

To perform a proof by induction, it is necessary to provide induction schemes. In our framework, these schemes are defined by a function, which, given a conjecture, selects the positions of variables where the induction will be applied (induction variables) and by a special set of terms called a test set which give us a finite description of the initial model of the rewrite system. Test sets are used to instantiate induction variables.

6.1 Induction Variables

Induction variables are variables which may trigger a rewriting step when instantiated. Given a conditional rewrite system R, to compute induction variables, we start by the computation of induction positions of function symbols. This computation is done only once and permits us to determine whether a variable of a term t is an induction variable or not. Then, it is not necessary to consult the axioms in order to select the induction variables of a conjecture. For example, if we consider the rewrite system $\{x + 0 \to x, x + s(y) \to s(x + y)\}$, then y is an induction variable of the term $x + y$, since the corresponding rewrite rules' left-hand sides $x + 0$ and $x + s(y)$ vary at the second position of $+$. We say also that 1 is an induction position[2] of $+$. The inductive positions allow us to restrict the instantiation of conjectures to a smaller set of inductive variables that trigger a rewriting step.

A sort S is *finitary* if the set of R-irreducible ground terms of sort S is finite. Otherwise, S is *infinitary*.

Definition 11 (Induction variables)**.** The set $\mathrm{IndVar}_R(c)$ of **induction variables** of a clause c with respect to a rewrite system R is the smallest subset of $\mathrm{Var}(c)$ satisfying the following conditions:

(1) if x is a variable of a finitary sort with respect to R then $x \in \mathrm{IndVar}_R(c)$;

(2) if $x = t|_p$, where t is a subterm of c of the form $f(t_1, \ldots, t_n)$, such that each t_i is a term built from constructors and variables, and p is an inductive position of f, then $x \in \mathrm{IndVar}_R(c)$.

If the rewrite system R is clear from the context, we denote the induction variables only by $\mathrm{IndVar}(c)$.

6.2 Test Sets

Let R be a left-linear conditional rewrite system. Let $\mathrm{dp}(R)$ be defined as the maximum depth of the left-hand sides of rules in R and $\mathrm{dp}_f(R)$ be defined as the maximum depth of the left-hand sides of rules in R with the root symbol f. Let R be a sufficiently complete and ground convergent conditional rewrite system. **Test sets** are computed for all defined functions f. The test set $\mathrm{TS}[f, s]$ for a defined function f and a sort s is the set of all constructor terms of depths smaller or equal to $\mathrm{dp}_f(R)$ with variables only on the deepest level. The test set TS is the union of all $\mathrm{TS}[f, s]$ for all defined functions f and all sorts s. A **test substitution** for a clause c instanciates induction variables of c by terms taken from a given test set whose variables are refreshed by renaming.

Example 12. Consider the rewrite system R with the rules

$$\begin{array}{ll} 0 + x \to x, & s(x) + y \to s(x + y), \\ \mathit{even}(0) \to \mathit{true}, & \mathit{even}(s(0)) \to \mathit{false}, \\ \mathit{even}(s^2(x)) \to \mathit{even}(x), & \\ \quad \mathit{even}(x) = \mathit{true} \Rightarrow \mathit{odd}(x) \to \mathit{false}, & \\ \quad \mathit{even}(x) = \mathit{false} \Rightarrow \mathit{odd}(x) \to \mathit{true} & \end{array}$$

where 0 and s are constructors of the sort *nat*, and $+$, *even*, and *odd* are defined functions. The final test set is equal to

$$\begin{aligned} \mathrm{TS} = \{&(+, \mathit{nat}, \{0, s(x)\}),\ (\mathit{even}, \mathit{nat}, \{0, s(0), s^2(x)\}), \\ &(\mathit{odd}, \mathit{nat}, \emptyset)\}. \end{aligned}$$

7 Inference System for Proofs by Induction

In this section we describe an inference system $\mathcal{I}$ (see Figure 1) for proving or disproving inductive consequences with respect to a given conditional rewrite system R.

The inference system operates on a tuple (E, H, G, T, S, B) where E is the set of conjectures, H is the set of induction hypotheses, G is the set of primal grammars, T is the list of new subgoals, S is the set of new signatures, and B is a Boolean flag.

[2]All positions start with 0 and proceed from left to right.

The SIMPLIFY inference rule allows us to perform the following tasks.

- Use of inductive rewriting to reduce a clause c with axioms from R, induction hypotheses from H, other conjectures from E that have not yet been proved and are smaller compared to c (which allows mutual simplification of conjectures), and primal grammars from G. We use a generalization of the complexity of equations given in [Bouhoula *et al.*, 1995] to compare clauses. This complexity allows, in particular, to handle non-orientable equations.
- Use of case rewriting.
- Application of positive decomposition taking advantage of the fact that constructors are free to simplify clauses.
- Delete redundant clauses or tautologies and application of occur check and negative clash if the constructors are free.
- Subsume clauses modulo the equational theory generated by Presburger rewrite systems.

The INDUCTIVE NARROWING inference rule allows us to derive new subgoals by the instantiation of induction variable of a conjecture by the elements of test sets and the simplification of the obtained instances. Notation $c :: T$ denotes pushing of c to the front of list T.

The definitions of inductive rewriting, case rewriting, positive decomposition, occur check, and negative clash can be found in [Bouhoula, 1997].

The PRIMAL GRAMMAR inference rule constructs a primal grammar from a supposed divergent sequence, and it generates new lemmas which will subsume all divergent sequences. Procedure REMOVELIST(T, E') removes any subgoal in T subsumed by a lemma in E'. Procedure PRIMGRAM(T) returns a new primal grammar G' with a new signature S' and new lemmas E'. If PRIMGRAM(T) fails, it returns the tuple $(\emptyset, \emptyset, \emptyset)$. The SUCCESS inference rule will be applied if the set of conjectures is exhausted. The inference rule DISPROOF applies if no other rule can be applied to c. The flag B prevents the PRIMAL GRAMMAR inference rule to be applied in an infinite loop.

The diagram in Figure 2 presents the strategy for applying the inference rules. SIMPLIFY$*$ means that the simplification rule is used exhaustively until no further simplifications can take place.

Given a set of conjectures, we start with the saturation of the application of the SIMPLIFY inference rule; if the resulting set of conjectures is exhausted, the SUCCESS inference rule is applied, confirming the validity of the conjectures in the initial model of R.

If the SIMPLIFY process does not exhaust the conjectures, we try to apply the INDUCTIVE NARROWING inference rule. If the application of INDUCTIVE NARROWING fails, we conclude that at least one of the conjectures is not valid in the initial model. Alternatively, if the INDUCTIVE NARROWING rule succeeds, we move to the simplification step. If, once again, the resulting set of conjectures is exhausted, the SUCCESS inference rule will be applied. However, if the simplification process does not exhaust the conjectures, we apply the PRIMAL GRAMMAR inference rule.

INIT: $\vdash (E, \emptyset, \emptyset, [], \emptyset, \mathit{false})$

SIMPLIFY: $(E \cup \{c\}, H, G, T, S, B) \vdash (E \cup E', H, G, T, S, B)$ **if** $E' = \mathit{simplification}(c, R, H, E, G)$

INDUCTIVE NARROWING: $(E \cup \{c\}, H, G, T, S, B) \vdash (E \cup E', H \cup \{c\}, G, c :: T, S, \mathit{true})$
where $E' = \bigcup_\sigma \mathit{simplification}(c\sigma, R, H, E, G)$ **and** σ is a test substitution of c **if** c has an induction variable

PRIMAL GRAMMAR: $(E, H, G, T, S, \mathit{true}) \vdash (E \cup E', H, G \cup G', T', S \cup S', \mathit{false})$
where $(G', S', E') = \text{PRIMGRAM}(T)$ **and** $T' = \text{REMOVELIST}(E', T)$

SUCCESS: $(\emptyset, H, G, T, S, B) \vdash \mathit{success}$

DISPROOF: $(E \cup \{c\}, H, G, T, S, B) \vdash \mathit{disproof}$ **if** no other rule applies to c

Figure 1: Inference system $\mathcal{I}$

If the application of the PRIMAL GRAMMAR rule succeeds, we generate new lemmas, add them to the set of conjectures, and repeat the process. If PRIMAL GRAMMAR fails, we iterate the process with the current set of conjectures.

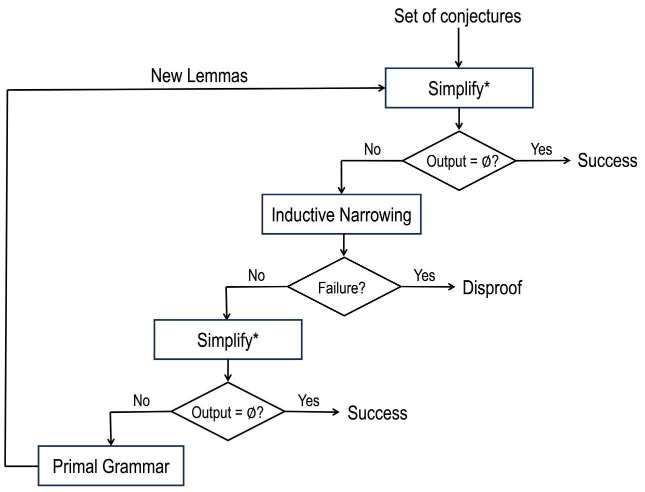

Figure 2: Strategy for applying inference rules

Proposition 13. *Let $E = \{c_0, c_1, c_2, \ldots\}$ be a primitive recursive set of clauses produced by the inference system $\mathcal{I}$. The sufficient length p of a prefix $E_p = \{c_0, \ldots, c_{p-1}\}$ to produce a primal grammar G_E schematizing E is 3.*

Remark. The inference process can generate multiple independent divergent sequences. In Proposition 13, the length p of a prefix E_p is, of course, associated with the length of the *dependent* divergent sequence $\{c_0, \ldots, c_{p-1}\}$.

We always consider fair derivations in this paper. Fairness roughly means that every clause in the set of conjectures E will be eventually modified by some inference rule. More formally, a derivation $(E_0, H_0, \cdots) \vdash (E_1, H_1, \cdots) \vdash \cdots$ is fair if either it ends by the application of DISPROOF or the set of persisting clauses $(\bigcup_i \bigcap_{j \geq i} E_j)$ is empty.

Our inference system $\mathcal{I}$ is sound and refutationally complete.

Theorem 14 (Soudness)**.** *Let R be a ground convergent rewrite system and let $(E_0, \emptyset, \emptyset, [], \emptyset, \mathit{false}) \vdash (E_1, H_1, G_1, T_1, S_1, B_1) \vdash \cdots$ be a fair derivation that does not end by the application of* DISPROOF*. Then the conjectures in E are valid in the initial model of R.*

Theorem 15 (Refutational Completeness)**.** *Let R be a ground convergent rewrite system which is strongly complete over free constructors. If $R \not\models E$, then all fair derivations issued from $(E, \emptyset, \emptyset, [], \emptyset, \mathit{false})$ terminate by the application of* DISPROOF.

8 Computer Experiments

Our method has been implemented in C++ with over 6000 lines of code and successfully proved automatically over fifty complex examples, including all the examples given in [Walsh, 1996], together with those left unresolved in that paper. These examples are the cornerstone for measuring the performance of a system dealing with divergent inductive proofs. These examples cannot be proved automatically with well-known theorem provers (e.g., ACL2, Isabelle, PVS, SPIKE, RRL, LEAN, Vampire) and related methods for handling divergence in proofs by induction. The systems performance of our prototype is highly dependent on the efficiency of the Primal Grammar part. In cases where only one divergent sequence is encountered for each application of Primal Grammar, the running time is always below 10ms, and usually around 5ms. If multiple divergent sequences are encountered simultaneously, the execution time may vary from 0.045s to 18.645s. A fairly complex example proving the correctness of insert sort, with numerous divergent sequences, requires 207 proof cycles (according to Figure 2), produces 33 lemmas, and runs in 6.05 seconds. The example taking the longest time, proving $(x * y)^z = x^z * y^z$, requires 56 proof cycles and runs in 18.645s. The mean execution time of all the examples is 0.62s and the standard deviation is 2.74s. All times are measured on a Dell Precision 3630 Tower with Intel Core i7-9700 $\times$ 8 processor and 16GB memory, running under Fedora Linux 39. These examples can be found at the github page https://github.com/BouhoulaHermann/IJCAI-2024.

Let us now observe how an inductive proof proceeds when it needs to construct and use primal grammars.

Example 16. Consider the rewrite system

$$\begin{array}{ll} x+0 \to x; & x+s(y) \to s(x+y); \\ x*0 \to 0; & x*s(y) \to (x*y)+x. \end{array}$$

of natural numbers with addition $+$ and multiplication $*$, and the conjecture $s(x)*y = y+(x*y)$. Despite the very simple specification, this proof triggers considerable complications with a tripple divergence and cannot be proved by other theorem provers.

Only y is an induction variable in the conjecture. Its test set is composed of the terms 0 and $s(x)$. The instantiation of y by 0 leads to a tautology. The inference system encounters first the divergent sequence of equations

$$\begin{array}{l} s((y+(x*y))+x) = s(y)+((x*y)+x), \\ s((s(y)+((x*y)+x))+x) = \\ \qquad s^2(y)+(((x*y)+x)+x), \\ s((s^2(y)+(((x*y)+x)+x))+x) = \\ \qquad s^3(y)+((((x*y)+x)+x)+x) \end{array}$$

out of which it constructs a primal grammar with the Presburger system

$$\hat{f}_0(0;y) \to y, \quad \hat{f}_0(n_0+1;y) \to s(\hat{f}_0(n_0;y)),$$
$$\hat{f}_1(0;x,y) \to x*y, \quad \hat{f}_1(n_1+1;x,y) \to \hat{f}_1(n_1;x,y)+x$$

and the lemma

$$\begin{array}{l} s((\hat{f}_0(n;y)+\hat{f}_1(n;x,y))+x) = \\ \quad s(\hat{f}_0(n;y))+(\hat{f}_1(n;x,y)+x). \end{array}$$

This lemma is then used to subsume the clause

$$\begin{array}{l} s((s^3(y)+((((x*y)+x)+x)+x))+x) = \\ \quad s^4(y)+(((((x*y)+x)+x)+x)+x). \end{array}$$

modulo the Presburger rewrite system using the counter substitution $n \leftarrow 3$. The inference system encounters then another divergent sequence of equations

$$s(s^2(0+y)+y) = s^2(0)+(s(0+y)+y)$$
$$s(s^3(0+y)+y) = s^2(0)+(s^2(0+y)+y)$$
$$s(s^4(0+y)+y) = s^2(0)+(s^3(0+y)+y)$$

out of which it constructs a primal grammar with the Presburger system

$$\hat{f}_3(0;y) \to 0+y, \quad \hat{f}_3(n+1;y) \to s(\hat{f}_3(n;y))$$

and the lemma

$$s(s^2(\hat{f}_3(n;y))+y) = s^2(0)+(s(\hat{f}_3(n;y))+y).$$

This lemma is then used to subsume the clause

$$s(s^5(0+y)+y) = s^2(0)+(s^4(0+y)+y)$$

with the counter substitution $n \leftarrow 3$. The inference system encounters yet another divergent sequence of equations

$$\begin{array}{l} s(s(s^2(0)+(s(0+y)+y))+y) = \\ \qquad s^3(0)+((s^2(0)+((0+y)+y))+y), \\ s(s^2(s^2(0)+(s^2(0+y)+y))+y) = \\ \qquad s^3(0)+(s(s^2(0)+(s(0+y)+y))+y), \\ s(s^3(s^2(0)+(s^3(0+y)+y))+y) = \\ \qquad s^3(0)+(s^2(s^2(0)+(s^2(0+y)+y))+y), \end{array}$$

out of which it constructs a primal grammar with the Presburger system

$$\begin{array}{l} \hat{f}_4(0,k;y) \to s^2(0)+(s(\hat{f}_3(k;y))+y), \\ \hat{f}_4(n+1,k;y) \to s(\hat{f}_4(n,k+1;y)), \\ \hat{f}_5(0,k;y) \to s^2(0)+(\hat{f}_3(k;y)+y), \\ \hat{f}_5(n+1,k;y) \to s(\hat{f}_5(n,k+1;y)), \end{array}$$

and the lemma

$$s(s(\hat{f}_4(n,0;y))+y) = s^3(0)+(\hat{f}_5(n,0;y)+y).$$

This lemma is then used to subsume the clause

$$\begin{array}{l} s(s^4(s^2(0)+(s(s^3(0+y))+y))+y) = \\ \quad s^3(0)+(s^3(s^2(0)+(s^3(0+y)+y))+y) \end{array}$$

with the counter substitution $n \leftarrow 3$. After this third subsumption there remain no conjectures in the inference system and the proof terminates with success.

Similarly, if we try to prove the conjecture $s(x)+x = s(x+x)$, the proof diverges and our system produces, after four cycles, a primal grammar with the Presburger system

$$\hat{f}_0(0;x) \to x, \qquad \hat{f}_0(n_0+1;x) \to s(\hat{f}_0(n_0;x)),$$
$$\hat{f}_1(0;x) \to x+x, \qquad \hat{f}_1(n_1+1;x) \to s(\hat{f}_1(n_1;x)),$$

and the lemma $s(s(\hat{f}_0(n;x)))+x = s(s(\hat{f}_1(n;x)))$ that allows breaking the divergence and finishing the proof.

Note that there is no solution in [Walsh, 1996] for finishing the proof of the first conjecture and two supplementary lemmas are necessary to prove the second one.

9 Concluding Remarks

We presented a new method that allows to perform inductive proofs in conditional theories which automatically detects divergence of proof traces and derives primal grammars together with new lemmas that schematize the divergent sequences and thus, allow breaking the divergence and finishing the proof. The main advantages of our method compared to previous methods are: (i) it is completely automatic, i.e., no interaction with the user, (ii) it preserves refutational completeness. Refutational completeness is particularly useful for detecting flaws in critical systems, and (iii) there is no risk of over-generalization.

Our method has been implemented in C++ and successfully proved several dozens of complex examples that fail with well-known theorem provers and related methods for capturing and schematizing divergence for proof by induction.

This research represents a significant advancement in automated reasoning as it can be integrated with existing automated and interactive proof systems to improve their performance. Moreover, computer experiments show that our method is very promising and can significantly reduce the time needed to verify critical systems.

We plan to generalize our method to automated theorem proving in membership equational logic [Bouhoula *et al.*, 2000; Bouhoula and Jouannaud, 2001], as well as in higher-order logic [Nipkow *et al.*, 2002].

References

[Aubin, 1976] Raymond Aubin. *Mechanizing structural induction*. PhD thesis, University of Edinburgh, UK, 1976.

[Baader and Nipkow, 1998] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[Basin and Walsh, 1992] David A. Basin and Toby Walsh. Difference matching. In D. Kapur, editor, *Proceedings 11th International Conference on Automated Deduction (CADE 1992), Saratoga Springs (New York, USA)*, volume 607 of *Lecture Notes in Computer Science*, pages 295–309. Springer, June 1992.

[Basin and Walsh, 1993] David A. Basin and Toby Walsh. Difference unification. In Ružena Bajcsy, editor, *Proceedings 13th International Joint Conference on Artificial Intelligence (IJCAI 1993), Chambéry (France)*, pages 116–122. Morgan Kaufmann, 1993.

[Bouhoula and Jouannaud, 1997] Adel Bouhoula and Jean-Pierre Jouannaud. Automata-driven automated induction. In *Proceedings 12th Annual IEEE Symposium on Logic in Computer Science (LICS 1997), Warsaw (Poland)*, pages 14–25. IEEE Computer Society, 1997.

[Bouhoula and Jouannaud, 2001] Adel Bouhoula and Jean-Pierre Jouannaud. Automata-driven automated induction. *Information and Computation*, 169(1):1–22, 2001. A preliminary version was published in [Bouhoula and Jouannaud, 1997].

[Bouhoula and Rusinowitch, 1993] Adel Bouhoula and Michaël Rusinowitch. Automatic case analysis in proof by induction. In Ružena Bajcsy, editor, *Proceedings 13th International Joint Conference on Artificial Intelligence (IJCAI 1993), Chambéry (France)*, pages 88—-94. Morgan Kaufmann, 1993.

[Bouhoula *et al.*, 1995] Adel Bouhoula, Emmanuel Kounalis, and Michaël Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.

[Bouhoula *et al.*, 2000] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.

[Bouhoula, 1997] Adel Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, 1997.

[Bouhoula, 2000] Adel Bouhoula. Simultaneous checking of completeness and ground confluence. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, (ASE 2000), Grenoble (France)*, pages 143–151. IEEE Computer Society, 2000.

[Bouhoula, 2009] Adel Bouhoula. Simultaneous checking of completeness and ground confluence for algebraic specifications. *ACM Transactions on Computational Logic*, 10(3):20:1–20:33, 2009. A preliminary version was published in [Bouhoula, 2000].

[Boyer and Moore, 1979] Robert S. Boyer and J Strother Moore. *A computational logic handbook*, volume 23 of *Perspectives in computing*. Academic Press, 1979.

[Bundy *et al.*, 1993] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[Chen *et al.*, 1990] Hong Chen, Jieh Hsiang, and Hwa-Chung Kong. On finite representations of infinite sequences of terms. In Stéphane Kaplan and Mitsuhiro Okada, editors, *Proceedings 2nd International Workshop on Conditional and Typed Rewriting Systems (CTRS 1990), Montreal (Canada)*, volume 516 of *Lecture Notes in Computer Science*, pages 100–114. Springer, 1990.

[Comon, 1995] Hubert Comon. On unification of terms with integer exponents. *Mathematical Systems Theory*, 28(1):67–88, 1995.

[de Moura and Ullrich, 2021] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Proceedings 28th International Conference on Automated Deduction (CADE 2021), Virtual Event*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[Dershowitz and Jouannaud, 1991] Nachum Dershowitz and Jean-Pierre Jouannaud. Notations for rewting. *Bulletin of EATCS*, 43:162–174, 1991.

[Hajdú *et al.*, 2020] Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with generalization in superposition reasoning. In Christoph Benzmüller and Bruce R. Miller, editors, *Proceedings 13th International Conference on Intelligent Computer Mathematics (CICM 2020), Bertinoro (Italy)*, volume 12236 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2020.

[Hermann and Galbavý, 1997] Miki Hermann and Roman Galbavý. Unification of infinite sets of terms schematized by primal grammars. *Theoretical Computer Science*, 176(1-2):111–158, 1997.

[Hozzová *et al.*, 2021] Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer induction in saturation. In André Platzer and Geoff Sutcliffe, editors, *Proceedings 28th International Conference on Automated Deduction (CADE 2021), Virtual Event*, volume 12699 of *Lecture Notes in Computer Science*, pages 361–377. Springer, 2021.

[Ireland and Bundy, 1996] Andrew Ireland and Alan Bundy. Extensions to a generalization critic for inductive proof. In Michael A. McRobbie and John K. Slaney, editors, *Proceedings 13th International Conference on Automated Deduction (CADE-13), New Brunswick (New Jersey, USA)*, volume 1104 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 1996.

[Kapur and Subramaniam, 1996] Deepak Kapur and Mahadevan Subramaniam. Lemma discovery in automated induction. In Michael A. McRobbie and John K. Slaney, editors, *Proceedings 13th International Conference on Automated Deduction (CADE-13), New Brunswick (New Jersey, USA)*, volume 1104 of *Lecture Notes in Computer Science*, pages 538–552. Springer, 1996.

[Kapur and Zhang, 1995] Deepak Kapur and Hantao Zhang. An overview of rewrite rule laboratory (RRL). *Computers & Mathematics with Applications*, 29(2):91–114, 1995.

[Kaufmann *et al.*, 2000] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[Rushby *et al.*, 1998] John M. Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[Salzer, 1992] Gernot Salzer. The unification of infinite sets of terms and its applications. In Andrei Voronkov, editor, *Proceedings 3rd International Conference on Logic Programming and Automated Reasoning (LPAR 1992), St. Petersburg (Russia)*, volume 624 of *Lecture Notes in Computer Science*, pages 409–420. Springer, 1992.

[Urso and Kounalis, 2004] Pascal Urso and Emmanuel Kounalis. Sound generalizations in mathematical induction. *Theoretical Computer Science*, 323(1-3):443–471, 2004.

[Walsh, 1996] Toby Walsh. Divergence critic for inductive proofs. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.