HAL
open science

Superposition with Structural Induction

Simon Cruanes

To cite this version:

Simon Cruanes. Superposition with Structural Induction. Clare Dixon; Marcelo Finger. Frontiers of Combining Systems, Springer, pp.172-188, 2017, 11th International Symposium on Frontiers of Combining Systems - FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings, 978-3-319-66166-7. 10.1007/978-3-319-66167-4_10 . hal-02062459

HAL Id: hal-02062459

https://inria.hal.science/hal-02062459v1

Submitted on 8 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Superposition with Structural Induction

Simon Cruanes

University of Lorraine, CNRS, Inria, LORIA, 54000 Nancy, France

Abstract. Superposition-based provers have been successfully used to discharge proof obligations stemming from proof assistants. However, many such obligations require induction to be proved. We present a new extension of typed superposition that can perform structural induction. Several inductive goals can be attempted within a single saturation loop, by leveraging AVATAR [1]. Lemmas obtained by generalization or theory exploration can be introduced during search, used, and proved, all in the same search space. We describe an implementation and present some promising results.

1 Introduction

Superposition-based theorem provers and SMT (Satisfiability Modulo Theory) solvers have considerably improved automation in some proof assistants thanks to *hammers* [2, 3]. However, because these proof assistants provide inductive datatypes, many theorems are out of reach of the automated provers, which are not able to perform inductive reasoning. Such theorems include basic properties of Peano arithmetic, reasoning about data structures such as lists and trees, manipulating syntax trees (which are often represented as a recursive datatype), etc.

Most state of the art theorem provers for first-order logic with equality are based on superposition [4–7]. However, they often lack support for types or (inductive) datatypes. Vampire [5] has recently gained some support for datatypes [8] but does not perform induction yet.

Automatic inductive provers do exist [9–11] but they are usually not complete (nor very efficient) on the classical first-order logic problems hammers rely on. INKA [10] was based on resolution, but not superposition. A recent extension to CVC4 [12] equips it with inductive reasoning, but so far no major superposition-based theorem prover has inductive capabilities. Kersani and Peltier modified Prover9 to handle induction [13], but only for natural numbers; it is unclear how their technique could be extended to arbitrary datatypes. Otter-λ [14] can use its (incomplete) higher-order unification algorithm to apply explicitly the induction principle, but it does not try to introduce any lemma nor does it handle defined functions or datatypes efficiently. Another superposition prover able to prove some inductive properties is Pirate [15, unpublished], but in its architecture each inductive property is solved in a separate saturation loop; it resembles more an inductive prover that would use a superposition prover for discharging subgoals.

In this work we propose a new architecture that permits a seamless integration of multiple induction attempts into the deduction process of a superposition prover. All the proof attempts are performed in the same saturation loop [4]. This has several advantages. First, if the problem does not actually need induction, a regular first-order proof can be obtained as usual. Second, once a particular inductive goal has been proved, it is considered as a normal first-order formula. This means it can participate in all the usual inference and simplification rules and contribute to the rest of the proof. Third, efforts are allocated to the various inductive proof attempts using the same clause selection heuristics that drive the first-order prover. This means the same elaborate heuristics can be reused for inductive proofs.

Our approach relies on a variant of superposition with polymorphic types, recursive functions, and inductive datatypes, as well as support for AVATAR [1] for reasoning by case (Sect. 3). Regular splitting without backtracking [16] could be used instead of AVATAR, but is less convenient and efficient. This richer variant of superposition treats defined functions efficiently. The prover can handle problems expressed in TIP [17] ("Thousands of Inductive Problems"). It encodes the non-first-order constructs of TIP, such as pattern matching, during a preprocessing step.

On top of this extension of superposition, we introduce a new rule to instantiate the structural induction schema and prove a property by induction (Sect. 4). Pursuing several inductive goals simultaneously is made possibly by introducing a cut rule on top of AVATAR (Sect. 5). The properties to prove by induction come from several sources: the input goal, explicit lemmas requested by the user, or subgoals needed in already ongoing proofs (Sect. 6). The numerous heuristics for guessing relevant lemmas that have been developed for decades [9,18–21] can be adapted to our framework. We also present a simple way to filter out invalid potential lemmas (Sect. 7).

To show the practical feasibility of the approach, we implemented the extensions of superposition and the inductive reasoning rules in Zipperposition, a modular prover (Sect. 8). Comparisons with CVC4 on the TIP benchmarks show that the implementation is reasonably competitive, and suggest that an implementation in E or Vampire could lead to excellent results.

2 Basic Definitions

We define some notions and notations that will be useful for the rest of the paper. An *atomic type* is a type constructor applied to 0 or more atomic types. A (polymorphic) *type* has the form $\Pi \alpha_1 \ \ldots \ \alpha_n.\ (\tau_1, \ldots, \tau_k) \to \tau$ where the α_i's are *type variables* and each τ_i is an atomic type. By s, t, u, v we denote *terms*, generated from variables x, y, z and *function symbols* f, g, h. By $\bar{t}$ we denote a finite (possibly empty) sequence of terms. A term is *ground* if it contains no variable. Given a term t and a *position* p, we write $t|_p$ for the subterm of t at p. We write $t \lhd u$ if t is a (strict) *subterm* of u, i.e. if there is a non-empty position p such that $t = u|_p$. A *substitution* is a mapping σ from variables to terms such

that the set of variables $\mathrm{dom}(\sigma) \stackrel{\mathrm{def}}{=} \{x \mid x \neq x\sigma\}$ is finite. We always implicitly restrict ourselves to well-typed terms, substitutions, etc.

A *literal* is an equation $s \simeq t$ or disequation $s \not\simeq t$. Note that $\simeq$ is a logical symbol, whereas $=$ denotes syntactic equality. A proposition p is implicitly represented by $p \simeq \top$. A *clause* is a disjunction of literals, denoted by C or D. The *empty clause* is the empty disjunction, equivalent to $\bot$ (false). We sometimes view clauses as multisets of literals. Ground literals and clauses are defined in the obvious way.

We reuse some concepts from AVATAR [1]. A *boolean mapping* $[\![\cdot]\!]$ is an injective mapping from clauses (and more generally, in our case, of formulas) into the propositional literals of a SAT solver. A *clause with assertion*, or *A-clause*, is a pair of a clause C and conjunction of boolean literals Γ, called the *trail*, and noted $C \leftarrow \Gamma$. It holds in an interpretation if either C holds, or Γ does not hold. We write $\sqcap$ for boolean conjunction and $\oplus$ for exclusive disjunction.

An *inductive (data)type* is defined by a set of *constructors*, at least one of which is non-recursive — we ignore mutually recursive datatypes, which can be encoded into a single datatype. A term t is *purely inductive* if every subterm of t whose type is inductive, has the form $c(t_1, \ldots, t_n)$ where c is a constructor symbol. A *constructor context* $C[\diamond]$ is a term built from constructors, function symbols of non-inductive type, and a unique occurrence of $\diamond$; *applying* the context to a term t, written $C[t]$, means replacing the occurrence of $\diamond$ by t. $C[t]$ is only defined if it is well typed. A Herbrand model is *standard* if (i) it satisfies the axioms of datatypes: exhaustiveness and disjointness of constructors; (ii) every (ground) term is equal to some purely inductive term; (iii) equivalence classes of inductive types are *acyclic*, i.e. for every non-trivial constructor context $C[\diamond]$, $t \not\simeq C[t]$ is true in the model.

We seek to establish the satisfiability of formulas in standard models. To achieve this, we will instantiate the *induction schema* over the inductive types. The induction schema for an inductive type τ is a second order formula parameterized by a variable $P : \tau \to \mathsf{bool}$, but we instantiate it into a first-order formula that will be (dis)proved by superposition.

Example 1 (Natural Numbers). The type of natural numbers, nat, is a classic inductive type whose constructors are $\{0, \mathsf{s}\}$. Its inductive values are all the natural numbers $\{0, \mathsf{s}(0), \ldots, \mathsf{s}^k(0), \ldots\}$. The induction schema is $\forall P : \mathsf{nat} \to \mathsf{bool}.\ P(0) \wedge (\forall n : \mathsf{nat}.\ P(n) \Rightarrow P(\mathsf{s}(n))) \Rightarrow \forall n.\ P(n)$.

Example 2 (Lists). The type of polymorphic lists is $\mathsf{list}(\alpha)$. Its constructors are $[\,]$ and $(::) : \alpha \times \mathsf{list}(\alpha) \to \mathsf{list}(\alpha)$. The purely inductive values of type $\mathsf{list}(\tau)$ are finite lists of purely inductive values of type τ. The induction schema on lists is

$$\forall \alpha.\ \forall P : \mathsf{list}(\alpha) \to \mathsf{bool}.\ P([\,]) \wedge (\forall x : \alpha\ l : \mathsf{list}(\alpha).\ P(l) \Rightarrow P(x :: l)) \Rightarrow \forall l.\ P(l)$$

3 Superposition with Recursive Functions and Datatypes

Before considering induction, we need the theorem prover to be able to handle problems that contain defined functions and datatypes. In addition, input prob-

lems can contain constructs that are outside of the realm of first-order terms, such as pattern matching and boolean conditionals ("if–then–else"). The solution we propose is multifold: (i) add some additional inference rules and simplification rules for datatypes; (ii) a notion of rewriting that does not rely on the term ordering; this is used to properly encode recursive functions; (iii) a preprocessing algorithm that removes non-first-order constructs by introducing newly defined functions and use them to encode the terms.

We adopt the notations and inference system of superposition from E [4] and AVATAR [1], and recall the following notions: an inference rule (noted with a single bar) infers the bottom clause(s) from the top clause(s); a simplification rule (with a double bar) *replaces* the top clause(s) with the bottom ones.

3.1 Recursive Functions and Rewriting

Superposition relies on a *term ordering* for orienting equations. This usually works well for first-order logic. However, recursive functions (on datatypes) are difficult to orient properly with such term orderings. Often, a rule $\forall \overline{x}.\ f(\overline{t}) \simeq u$ defining f will be oriented right-to-left because u will contain several occurrences of variables, be heavier (in KBO), etc. whereas we would like it to be oriented in the more natural left-to-right direction.

To unfold recursive functions efficiently, we translate them into rewrite rules. By construction, the left-hand side of the rewrite rules that define f is of the form $f(t_1, \ldots, t_n)$ where the t_i are generated from variables and constructors. Rules are also non-overlapping, ensuring that the resulting rewrite system is confluent: at most one rule will apply to any tuple of arguments. Rewriting is done left-to-right, regardless of the term ordering. This strategy is not complete in general,[1] but it guarantees that unfolding a function definition acts as an efficient simplification step. Recursive predicates are defined analogously, using rules of the form $l \rightsquigarrow C_i$ (rewriting a positive literal into a set of clauses) and $\neg l \rightsquigarrow \bigwedge_i C_i$ (rewriting a negative literal into a set of clauses). This sort of rewriting is known as *deduction modulo* [22] and, in our context, can be expressed as *polarized resolution* [23].

Having a well-delimited set of rules that define a function also enables the notion of argument position, defined below. This notion is useful because it provides some insight on which arguments influence the control flow of the function, and which ones are just carried around passively or serve as accumulators. In particular, it is pointless to try to perform induction on a passive argument, because the constructors at such positions cannot be eliminated.

Definition 1 (Argument Positions). Given a function f with k arguments, defined by rules

$$f(t_{1,1}, \ldots, t_{1,k}) \rightsquigarrow u_1, \ldots, f(t_{n,1}, \ldots, t_{n,k}) \rightsquigarrow u_n$$

[1] It might even induce rewriting loops in some cases where the term ordering used by superposition and the rewrite system are not compatible. In our experience this does not seem to happen often.

we say that each i, for $1 \leq i \leq k$, is an *argument position* of f. An argument position i is *passive* if every occurrence of f in $(u_j)_{j=1}^n$ has $t_{i,j}$ as i th argument; in other words, if f always calls itself with the same i th argument. A non-passive argument position i is an *accumulator* if every $(t_{i,j})_{j=1}^n$ is a variable; it is *primary* otherwise. Intuitively, a primary position is one that the function might examine for determining whether to recurse or not.

Example 3. (a) $+$ with the definition $0 + x \rightsquigarrow x, \mathsf{s}(x) + y \rightsquigarrow \mathsf{s}(x + y)$: the first argument is primary and the second one, passive. (b) $\leq$ defined by $(0 \leq x) \rightsquigarrow \top, (s(x) \leq 0) \rightsquigarrow \bot, (s(x) \leq s(y)) \rightsquigarrow (x \leq y)$: both positions are primary. (c) qrev with the rules $\mathsf{qrev}([\,], x) \rightsquigarrow x, \mathsf{qrev}(x :: y, z) \rightsquigarrow \mathsf{qrev}(y, x :: z)$: the first position is primary, the second is an accumulator.

3.2 Preprocessing the Input

Our prover can parse problems expressed in TIP [17], an extension of SMT-LIB [24] with recursive functions, polymorphism, and datatypes. However, many constructs in this language have no straightforward equivalent in a superposition prover, in which there are only clauses and first-order terms. These constructs are pattern matching, conditionals ("if–then–else"), lambda abstractions, and let-bindings; let-bindings are expanded, conditionals and pattern matches are either named, or become toplevel case distinction as a set of rewrite rules. Again, the rewrite rules generated by our encoding are terminating and confluent; they are also orthogonal to the other rewrite rules because their head symbol is a fresh constant. We show a few examples of encodings in Figure 1.

3.3 Inference Rules for Constructors

We consider the algebra of freely generated datatypes, such as Peano numbers, lists, or binary trees. This fragment is general enough to express many classic types and data structures, yet it is reasonably simple. Other theories such as rational arithmetic can also be used (e.g. using Hierarchic superposition [25]) but no induction will be performed on variables of these types.

Even without considering induction, datatypes need dedicated inference rules to account for acyclicity; other properties such as injectivity can be accounted for by adding either rules or axioms. Some SMT solvers have decision procedures for datatypes [26–28]. Similar work exists for superposition [8, 15, 29]. We use a small set of rules, as presented in Figure 2. In the rules, c and c' are distinct inductive constructors (e.g., the empty list $[\,]$, the successor symbol, etc.). The positive version of $\mathsf{Acyclicity}$ rule can also be used as a simplification when the unifier σ is trivial. These rules are sound with respect to standard models, but do not, by themselves, ensure completeness without induction.

4 Proving Formulas by Induction

Let us first look at a single proof by induction before we consider how to integrate such proofs in the superposition machinery (Sect. 5). An *inductive goal* is a closed

original	encoded
(declare−datatype Nat ((z) (s Nat)))	
(define−fun−rec leq ((x Nat)(y Nat)) Bool	$\forall x.\ \mathsf{leq}(z, x) \rightsquigarrow \top$
(match x (**case** z true)	$\forall x.\ \mathsf{leq}(\mathsf{s}(x), z) \rightsquigarrow \bot$
(**case** (s x2)	$\forall x\ y.\ \mathsf{leq}(\mathsf{s}(x), \mathsf{s}(y)) \rightsquigarrow \mathsf{leq}(x, y)$
(match y (**case** z false)	
(**case** (s y2) (leq x2 y2))))))	
(define−fun pred ((x Nat)) Nat	$\mathsf{pred}(z) \rightsquigarrow z$
(match x	$\forall x.\ \mathsf{pred}(\mathsf{s}(x)) \rightsquigarrow x$
(**case** z z)	$\forall x.\ \mathsf{fact}(x) \rightsquigarrow \mathsf{f}(x, \mathsf{leq}(x, \mathsf{s}(z)))$
(**case** (s x2) x2)))	$\forall x.\ \mathsf{f}(x, \top) \rightsquigarrow \mathsf{s}(z)$
(define−fun−rec fact ((x Nat)) Nat	$\forall x.\ \mathsf{f}(x, \bot) \rightsquigarrow \mathsf{mult}(x,$
(**let** ((one (s z)))	$\mathsf{f}(\mathsf{pred}(x),$
(if (leq x one)	$\mathsf{leq}(\mathsf{pred}(x), \mathsf{s}(z)))$
one	where f is fresh
(mult x (fact (pred x)))))	
(declare−fun g (Nat Nat) Nat)	
(define−fun h ((x Nat) (y Nat)) Bool	$\forall x\ y.\ \mathsf{h}(x, y) \rightsquigarrow x \simeq y \Rightarrow \mathsf{f}_2(y, x) \simeq \mathsf{f}_2(y, y)$
(**let**	$\forall y\ z.\ \mathsf{f}_2(y, z) \rightsquigarrow \mathsf{g}(y, \mathsf{s}(z))$
((g2	where f_2 is fresh
(lambda ((x Nat)) (g y (s x)))))	
(=> (= x y) (= (g2 x) (g2 y)))))	

Fig. 1: Encoding a few expressions and definitions

formula $\forall x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m.\ \bigwedge_j C_j$ where the variables x_i have inductive types and each C_j is a clause. To try and prove such a goal, we instantiate the structural induction schema over a non-empty subset of $\overline{x}$ into a first-order formula F, and try to refute $\neg F$ by the usual process of Skolemizing variables, reducing $\neg F$ to conjunctive normal form (CNF), and performing inferences until $\bot$ is deduced. We can instantiate the induction schema for a goal $\forall x_1, \ldots, x_k.\ G$ on the variables $x_1, x_2, \ldots, x_k$ ($k \geq 1$) by instantiating it on x_1 first (taking $P(x) = \forall x_2\ \ldots\ x_k.\ G$), obtaining F, and then by applying the induction principle on $x_2, \ldots, x_k$ to every occurrence of P in F.

4.1 Instantiating the Induction Schema

Before we explain how to instantiate the induction schema over a given set of variables, we must first define the notions of *inductive Skolem constant* and *coverset*.

Definition 2 (Coverset [30]). A *coverset* for an inductive type τ is a set of terms built from inductive constructors and variables $x_1, \ldots, x_n$ such that each

$$\frac{c(\overline{t}) \simeq c'(\overline{t'}) \vee D}{D}\ \textsf{Disjointness+} \qquad \frac{c(\overline{t}) \not\simeq c'(\overline{t'}) \vee D}{\top}\ \textsf{Disjointness-}$$

$$\frac{c(t_1, \ldots, t_n) \simeq c(t'_1, \ldots, t'_m) \vee D}{\bigwedge_{i=1}^{n} (t_i \simeq t'_i \vee D)}\ \textsf{Injectivity}$$

$$\frac{t \simeq C[u] \vee D \qquad t\sigma = u\sigma}{D\sigma}\ \textsf{Acyclicity+} \qquad \frac{t \not\simeq C[t] \vee D}{\top}\ \textsf{Acyclicity-}$$

where $C[\diamond]$ is a non-trivial constructor context

Fig. 2: Inference rules to deal with inductive constructors

variable x_i occurs in exactly one position, and $\forall t : \tau.\ \bigoplus_{u \in S} \exists x_1\ \ldots\ x_n.\ t \simeq u$ is valid in standard models. The terms of a coverset are distinct in any model.

Definition 3 (Inductive Skolem constant). An *inductive Skolem constant* $\mathfrak{i}$ is a Skolem constant of inductive type.

Definition 4 (Ground Coverset). A *ground coverset* $\kappa(\mathfrak{i})$ for an inductive Skolem constant $\mathfrak{i}$ is a set of ground terms obtained by replacing all variables in a coverset with fresh Skolem constants, such that $\bigoplus_{t \in \kappa(\mathfrak{i})} \mathfrak{i} \simeq t$ holds in any model. The elements of $\kappa(\mathfrak{i})$ represent all the possible "shapes" of $\mathfrak{i}$ in any model. If $t, \mathfrak{i} : \tau$ and there is some $t' \in \kappa(\mathfrak{i})$ such that $t \lhd t'$, we write $\mathrm{sub}(t, \mathfrak{i})$. We define $\kappa_{\downarrow}(\mathfrak{i}) = \{t \in \kappa(\mathfrak{i}) \mid \exists t' \lhd t.\ \mathrm{sub}(t', \mathfrak{i})\}$ — the set of recursive cases.

Example 4. The coversets of the type nat from Example 1 are of the form $\{0, \mathsf{s}(0), \ldots, \mathsf{s}^k(0), \mathsf{s}^{k+1}(x)\}$ for some $k \geq 0$.

Example 5. The coversets of the type $\mathsf{list}(\tau)$ from Example 2 are of the form $\{[\,], x_1 :: [\,], \ldots, x_1 :: \ldots :: x_m :: y\}$ where $m > 0$, $x_1, \ldots, x_m : \tau$, and $y : \mathsf{list}(\tau)$.

To prove an inductive goal $F \stackrel{\text{def}}{=} \forall x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m.\ \bigwedge_i C_i$ by induction over variables $x_1, \ldots, x_n$, we start by skolemizing each x_i with $\mathfrak{i}_i$ and each y_j with $\mathfrak{c}_j$. Then, we map each inductive Skolem constant $\mathfrak{i}_i$ to a ground cover set $\kappa(\mathfrak{i}_i)$. Our objective is to refute the following set of clauses:

$$\begin{aligned}
&\textstyle\bigcup_{t_1 \in \kappa(\mathfrak{i}_1), \ldots, t_n \in \kappa(\mathfrak{i}_n)} \left(\mathrm{cnf}(\neg\left(\bigwedge_i C_i[\overline{x} \mapsto \overline{t}, \overline{y} \mapsto \overline{\mathfrak{c}}]\right)) \leftarrow \textstyle\prod_{j=1}^{n} [\![\mathfrak{i}_j \simeq t_j]\!]\right) \\
&\cup \textstyle\bigcup_{u_1 \in \kappa_{\downarrow}(\mathfrak{i}_1), \ldots, u_n \in \kappa_{\downarrow}(\mathfrak{i}_n)} \left(\bigcup_i \left\{\forall \overline{y}.\ C_i[\overline{x} \mapsto \overline{u}] \leftarrow \textstyle\prod_{j=1, \mathrm{sub}(t_i, u_i)}^{n} [\![\mathfrak{i}_j \simeq t_j]\!]\right\}\right) \\
&\cup \textstyle\bigcup_{j=1}^{n} \left\{\bigoplus_{t \in \kappa(\mathfrak{i}_j)} [\![\mathfrak{i}_j \simeq t]\!]\right\}
\end{aligned}$$

The first set of clauses comes from the negation of our goal, after skolemization and case split (using a coverset to examine the possible shapes of these Skolem constants $(\mathfrak{i}_j)_j$). The second set comes from inductive hypothesis: to prove $\bot$ from $\neg(\bigwedge_i C_i)[x \mapsto n_0]$, in the case $n_0 \simeq \mathsf{s}(n_1)$, we need the hypothesis $(\bigwedge_i C_i)[x \mapsto n_1]$. The third set of boolean formulas, sent to the SAT solver, forces each Skolem constant to be equal to exactly one member of its ground coverset. We recall that $[\![\cdot]\!]$ turns a literal or clause into a propositional atom.

Example 6 (Associativity of +). Let $F \stackrel{\text{def}}{=} \forall x\ y\ z : \mathsf{nat}.\ x + (y + z) \simeq (x + y) + z$. To prove F, we perform induction on $\{x\}$, with Skolem symbols $\{\mathrm{x}_0, \mathrm{y}_0, \mathrm{z}_0\}$ and ground coverset $\kappa(\mathrm{x}_0) = \{0, \mathsf{s}(\mathrm{x}_1)\}$. The resulting clauses are:

$$
\begin{gathered}
0 + (\mathrm{y}_0 + \mathrm{z}_0) \not\simeq (0 + \mathrm{y}_0) + \mathrm{z}_0 \leftarrow [\![\mathrm{x}_0 \simeq 0]\!] \\
\mathsf{s}(\mathrm{x}_1) + (\mathrm{y}_0 + \mathrm{z}_0) \not\simeq (\mathsf{s}(\mathrm{x}_1) + \mathrm{y}_0) + \mathrm{z}_0 \leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \\
\forall y\ z.\ \mathrm{x}_1 + (y + z) \simeq (\mathrm{x}_1 + y) + z \leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \\
[\![\mathrm{x}_0 \simeq 0]\!] \oplus [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]
\end{gathered}
$$

Now, superposition (and AVATAR) can prove $\bot$ from these clauses:

1 induction(F).base	$0 + (\mathrm{y}_0 + \mathrm{z}_0) \not\simeq (0 + \mathrm{y}_0) + \mathrm{z}_0 \leftarrow [\![\mathrm{x}_0 \simeq 0]\!]$
2 def(+)	$0 + x \simeq x$
3 rewrite(1,2)	$\mathrm{y}_0 + \mathrm{z}_0 \not\simeq \mathrm{y}_0 + \mathrm{z}_0 \leftarrow [\![\mathrm{x}_0 \simeq 0]\!]$
4 eq-res(3)	$\bot \leftarrow [\![\mathrm{x}_0 \simeq 0]\!]$
5 avatar(4)	$\neg[\![\mathrm{x}_0 \simeq 0]\!]$
6 induction(F).hyp	$\forall y\ z.\ \mathrm{x}_1 + (y + z) \simeq (\mathrm{x}_1 + y) + z \leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$
7 induction(F).rec	$\mathsf{s}(\mathrm{x}_1) + (\mathrm{y}_0 + \mathrm{z}_0) \not\simeq (\mathsf{s}(\mathrm{x}_1) + \mathrm{y}_0) + \mathrm{z}_0 \leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$
8 def(+)	$\mathsf{s}(x) + y \simeq \mathsf{s}(x + y)$
9 rewrite(7,8)	$\mathsf{s}(\mathrm{x}_1 + (\mathrm{y}_0 + \mathrm{z}_0)) \not\simeq \mathsf{s}((\mathrm{x}_1 + \mathrm{y}_0) + \mathrm{z}_0) \leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$
10 sup(6,9)	$\mathsf{s}(\mathrm{x}_1 + (\mathrm{y}_0 + \mathrm{z}_0)) \not\simeq \mathsf{s}(\mathrm{x}_1 + (\mathrm{y}_0 + \mathrm{z}_0)) \leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$
11 eq-res(10)	$\bot \leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$
12 avatar(11)	$\neg[\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$
13 induction(F).case-split	$[\![\mathrm{x}_0 \simeq 0]\!] \sqcup [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$
14 res(5,12,13)	$\bot$

The last inference is done by the SAT solver after the addition of the boolean constraints $\neg[\![\mathrm{x}_0 \simeq 0]\!]$ and $\neg[\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!]$, establishing unsatisfiability.

Example 7 (Transitivity of $\leq$). Let $F \stackrel{\text{def}}{=} \forall x\ y\ z : \mathsf{nat}.\ x \leq y \wedge y \leq z \Rightarrow x \leq z$, where $\leq$ is defined by the rules $\{\forall x.\ 0 \leq x, \forall x.\ \neg(s(x) \leq 0), \forall x\ y.\ x \leq y \Longleftrightarrow s(x) \leq s(y)\}$. To prove F, we perform induction on $\{x, y, z\}$, with Skolem symbols $\{\mathrm{x}_0, \mathrm{y}_0, \mathrm{z}_0\}$ and ground coversets $\kappa(\mathrm{x}_0) = \{0, \mathsf{s}(\mathrm{x}_1)\}$, $\kappa(\mathrm{y}_0) = \{0, \mathsf{s}(\mathrm{y}_1)\}$, $\kappa(\mathrm{z}_0) = \{0, \mathsf{s}(\mathrm{z}_1)\}$. We obtain the following set of clauses, which is first-order

refutable:

$$
\begin{aligned}
0 \leq 0 &\leftarrow [\![\mathrm{x}_0 \simeq 0]\!] \sqcap [\![\mathrm{y}_0 \simeq 0]\!] \\
0 \leq 0 &\leftarrow [\![\mathrm{y}_0 \simeq 0]\!] \sqcap [\![\mathrm{z}_0 \simeq 0]\!] \\
\mathsf{s}(\mathrm{x}_1) \leq 0 &\leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \sqcap [\![\mathrm{y}_0 \simeq 0]\!] \\
\mathsf{s}(\mathrm{x}_1) \leq \mathsf{s}(\mathrm{y}_1) &\leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \sqcap [\![\mathrm{y}_0 \simeq \mathsf{s}(\mathrm{y}_1)]\!] \\
\mathsf{s}(\mathrm{y}_1) \leq 0 &\leftarrow [\![\mathrm{y}_0 \simeq \mathsf{s}(\mathrm{y}_1)]\!] \sqcap [\![\mathrm{z}_0 \simeq 0]\!] \\
\mathsf{s}(\mathrm{y}_1) \leq \mathsf{s}(\mathrm{z}_1) &\leftarrow [\![\mathrm{y}_0 \simeq \mathsf{s}(\mathrm{y}_1)]\!] \sqcap [\![\mathrm{z}_0 \simeq \mathsf{s}(\mathrm{z}_1)]\!] \\
\neg(0 \leq 0) &\leftarrow [\![\mathrm{x}_0 \simeq 0]\!] \sqcap [\![\mathrm{z}_0 \simeq 0]\!] \\
\neg(\mathsf{s}(\mathrm{x}_1) \leq 0) &\leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \sqcap [\![\mathrm{z}_0 \simeq 0]\!] \\
\neg(\mathsf{s}(\mathrm{x}_1) \leq \mathsf{s}(\mathrm{z}_1)) &\leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \sqcap [\![\mathrm{z}_0 \simeq \mathsf{s}(\mathrm{z}_1)]\!] \\
\neg(0 \leq \mathsf{s}(\mathrm{z}_1)) &\leftarrow [\![\mathrm{x}_0 \simeq 0]\!] \sqcap [\![\mathrm{z}_0 \simeq \mathsf{s}(\mathrm{z}_1)]\!] \\
\neg(\mathrm{x}_1 \leq \mathrm{y}_1) \vee \neg(\mathrm{y}_1 \leq \mathrm{z}_1) \vee (\mathrm{x}_1 \leq \mathrm{z}_1) &\leftarrow [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \sqcap [\![\mathrm{y}_0 \simeq \mathsf{s}(\mathrm{y}_1)]\!] \sqcap [\![\mathrm{z}_0 \simeq \mathsf{s}(\mathrm{z}_1)]\!] \\
& [\![\mathrm{x}_0 \simeq 0]\!] \sqcup [\![\mathrm{x}_0 \simeq \mathsf{s}(\mathrm{x}_1)]\!] \\
& [\![\mathrm{y}_0 \simeq 0]\!] \sqcup [\![\mathrm{y}_0 \simeq \mathsf{s}(\mathrm{y}_1)]\!] \\
& [\![\mathrm{z}_0 \simeq 0]\!] \sqcup [\![\mathrm{z}_0 \simeq \mathsf{s}(\mathrm{z}_1)]\!]
\end{aligned}
$$

Remark 1. Using AVATAR for keeping track of case splits allows an inductive lemma, once proved, to become a normal axiom and participate in other proofs using the following simplifications:

$$
\frac{C \leftarrow \Gamma \sqcap a}{C \leftarrow \Gamma}\ \mathsf{AvatarSimp+} \qquad \frac{C \leftarrow \Gamma \sqcap \neg a}{\top}\ \mathsf{AvatarSimp\text{-}}
$$

if the SAT-solver has proved a (propagated at level 0)

4.2 Selecting the Induction Variables

We have seen how to prove an inductive goal by induction over a set of variables. But how do we know which variables to choose? Going back to the case of Example 6 (associativity of addition), there are three variables, but only the leftmost one will lead to a successful induction.

The relevant notion here is that of *primary positions* under functions (Definition 1). Here we draw inspiration from Aubin's work [21]. The heuristic is that if a variable appears in at least one *primary occurrence*, it is a candidate for induction. Let $G \stackrel{\text{def}}{=} \forall \overline{x}\ \overline{y}.\ \bigwedge_i C_i$ be an inductive goal, with variables $\overline{x}$ having an inductive type. A *primary occurrence* in G is a position p in some clause C_i such that (i) $C_i|_p$ does not occur directly under a constructor, and (ii) every non-empty prefix of p occurs either below a constructor, an uninterpreted symbol, or under a primary argument position of a defined function or predicate. Intuitively, a subterm (such as a variable on which we might do induction) is a primary occurrence in G if replacing this subterm by a constructor-headed term has a chance of making G reducible by some rewrite rule. In Example 6, the goal is $\forall x\ y\ z : \mathsf{nat}.\ x + (y + z) \simeq (x + y) + z$. Of all three variables, only x appears in primary occurrences.

Now consider Example 7, where the goal is $\forall x\ y\ z : \mathsf{nat}.\ x \leq y \wedge y \leq z \Rightarrow x \leq z$. All variables occur in primary positions, because both positions of $\leq$ are primary. However, simply replacing x with a constructor-headed term will not always suffice to reduce the (ground) goal negation; indeed $\mathsf{s}(\mathrm{x}_1) \leq \mathrm{y}_0$ and

$\neg(\mathsf{s}(\mathrm{x}_1) \leq \mathrm{z}_0)$ cannot be reduced. Therefore, we extend Aubin's heuristic [21]. If two variables in primary positions in the goal G occur immediately under the same defined symbol, both at primary argument positions, then we perform induction on both of them simultaneously. For Example 7 that means that we perform induction over $\{x, y, z\}$, which succeeds.

5 Performing Several Inductive Proofs with AVATAR

In practice, we need to carry out several proofs by induction. It is necessary when such a proof depends on other properties that are themselves proved by induction. One such case is nested induction: to prove $\forall x\ y.\ x+y \simeq y+x$, the lemmas $\forall x.\ x+0 \simeq x$ and $\forall x\ y.\ x+\mathsf{s}(y) \simeq \mathsf{s}(x+y)$ are required. We wish to carry out all these proofs within a single saturation loop of the superposition prover, to reuse the existing algorithms and main loop. Fortunately, AVATAR makes it easy to introduce several *lemmas* and interleaving their proof with the main saturation process. Given a (candidate) lemma F (a closed first-order formula), the clauses $\{C \leftarrow [\![\mathsf{lemma}\ F]\!] \mid C \in \mathrm{cnf}(F)\} \cup \{D \leftarrow \neg[\![\mathsf{lemma}\ F]\!] \mid D \in \mathrm{cnf}(\neg F)\}$ are added to the saturation set. This corresponds to a boolean split over $F \vee \neg F$, where the choice between F and $\neg F$ is represented by the boolean valuation of the propositional literal $[\![\mathsf{lemma}\ F]\!]$.

Definition 5 (Lemma Introduction). The *introduction rule* of a lemma F, where F is a first-order formula, is the following inference rule:

$$\frac{\top}{\begin{array}{l}\bigwedge_{C\in\mathrm{cnf}(F)} C \leftarrow [\![\mathsf{lemma}\ F]\!] \\ \wedge \bigwedge_{D\in\mathrm{cnf}(\neg F)} D \leftarrow \neg[\![\mathsf{lemma}\ F]\!]\end{array}}\ \mathsf{Lemma}$$

Theorem 1. *The inference rule* Lemma *is sound.*

Proof. Lemma is similar to an AVATAR boolean split on $F \vee \neg F$ using the boolean $[\![F]\!]$ (F, being closed, is either valid or unsatisfiable). Since $[\![\neg F]\!] \stackrel{\text{def}}{=} \neg[\![F]\!]$, we obtain the trivial constraint $[\![F]\!] \sqcup \neg[\![F]\!]$ and the "A-formulas" $F \leftarrow [\![F]\!]$ and $\neg F \leftarrow \neg[\![F]\!]$ that can then be reduced to CNF. In essence, Lemma is using an adaptation of AVATAR splitting to formulas of the form $F \vee \neg F$ where F is closed.

In a part of the search space, inference with A-clauses of the form $C \leftarrow [\![F]\!]$ will correspond to using the lemma F, assuming it has been proved; in another part, inferences with A-clauses of the form $D \leftarrow \neg[\![F]\!]$ will possibly lead to (conditional) proofs of F by reaching clauses of the form $\bot \leftarrow \neg[\![F]\!] \sqcap \Gamma$ (proof of F under assumptions $\neg\Gamma$).

Remark 2 (Fairness and Lemmas). Using Lemma on a non-theorem formula F does not prevent an unsatisfiable combined state from being reached. The proof of each lemma is interleaved with the rest of the saturation process. Thanks to this, it is possible to introduce several (candidate) lemmas even if they are not all true or provable. However, it might take a longer time to find a solution, because of the larger search space.

6 Finding Subgoals and Lemmas by Generalization

In this section, we examine several ways of guessing new inductive goals that are likely to help existing proof attempts progress. There is a large amount of literature dedicated to lemma guessing, either by generalizing a subgoal [9, 15, 20, 21] or by exploring an equational theory systematically to find formulas that *seem* to hold based on testing [12, 31]. In this paper we present simple, relatively straightforward techniques that already yield good results; more sophisticated heuristics can be added on top.

Even though lemmas can either be proved on the fly by introducing a cut, or provided as axioms in the input file, they will be used in the same way in both cases. A subgoal is never generalized and replaced by a lemma; rather, we introduce a lemma which, if proved, will solve the subgoal by regular superposition.

6.1 Proving Subgoals by Induction

A superposition prover starts by reducing the input problem in CNF, in our case with some additional transformations (Sect. 3.2). A clause C containing at least one inductive Skolem constant can be negated, the constants replaced by fresh variables, resulting in an inductive goal that can be tested and then proved.

Similarly, during the course of saturation with some induction attempts, clauses of the form $C \leftarrow \neg[\![\mathsf{lemma}\ G]\!] \sqcap \Gamma$ (where G is an inductive goal) are clauses that need to be reduced to $\bot$ if G is to be proved. Again, if C contains at least one inductive Skolem constant, we can negate it, replace constants by variables, and assess the resulting goal.

More precisely, to prove a goal $\bigwedge_i C_i$ by generalizing inductive Skolem constants $\mathsf{i}_1, \ldots, \mathsf{i}_n$ ($n \geq 1$) occurring in the clauses C_i: (i) we replace each i_j by a fresh variable x_j; (ii) we negate every literal in the C_i and swap conjunctions and disjunctions; (iii) we redistribute conjunction over disjunction to get back to a CNF. Clauses in the result are quantified over $\{x_j\}_j$ and a subset of $\biguplus_i \mathrm{freevars}(C_i)$. This generalization technique also applies to the original goal after it has been negated and reduced to CNF during preprocessing. It makes it possible to prove inductively goals that are not in CNF.

A subtlety here is that if an induction variable j is a sub-case of some other constant ($\mathsf{j} \in \kappa_{\downarrow}(\mathsf{i})$), there exist induction hypotheses (in other clauses) that might be needed for nested induction. In this case we also try the inductive goal where j is *not* replaced by a fresh variable, and run both proof attempts simultaneously.[2]

6.2 Generalizing subgoals

An inductive goal G might not be provable by induction directly. For example, doing induction on x to prove $\forall x.\ x + (x + x) \simeq (x + x) + x$ will not succeed: in

[2] Our framework allows attempting to prove several distinct inductive goals to solve a single subgoal.

the recursive case $\mathrm{x}_0 = \mathsf{s}(\mathrm{x}_1)$, the clauses are

$$\begin{gathered}\mathrm{x}_1 + (\mathrm{x}_1 + \mathrm{x}_1) \simeq (\mathrm{x}_1 + \mathrm{x}_1) + \mathrm{x}_1 \\ \mathsf{s}(\mathrm{x}_1) + (\mathsf{s}(\mathrm{x}_1) + \mathsf{s}(\mathrm{x}_1)) \not\simeq (\mathsf{s}(\mathrm{x}_1) + \mathsf{s}(\mathrm{x}_1)) + \mathsf{s}(\mathrm{x}_1)\end{gathered}$$

but even after reduction, the hypothesis cannot rewrite the negative clause in any way because of successor symbols that appeared at passive positions. Following, once more, Aubin [21], we generalize *the primary occurrences* of a variable in a goal if it occurs at least twice in primary positions, and at least once in passive positions. In this way, doing inference on the primary occurrences will have better chances of succeeding. This generalization is only performed if the generalized goal still passes tests successfully (see Sect. 7).

Similarly, if a non-variable, non constructor-headed term occurs at least twice in primary positions, and is neither a variable nor constructor-headed, we can generalize it the same way.

Heuristics for guessing relevant lemmas from a goal have been developed for decades [9, 18–21] and can be adapted to our framework. For those that require to examine both the current subgoal and the induction hypothesis, more bookkeeping would be needed, because these objects live in the unstructured set of clauses, rather than in a sequent.

Remark 3. Speculating lemmas can be detrimental to the search space, by introducing many new clauses and performing arbitrary cuts. Therefore, the application of this rule must be heuristically restricted. In our implementation, we forbid deeply nested applications of generalization (beyond a small, user definable limit). Developing more advanced heuristics is however necessary.

7 Testing Conjectures before trying to prove them

Heuristics, as useful as they are, can mislead a solver into trying to prove inductive goals that are not valid. Attempting to prove an invalid goal with likely lead to a non-terminating superposition saturation on its own, draining memory and CPU resources away from the main proof effort. It pays to perform some limited amount of computation to try and rule out invalid goals.

To test a goal $G \stackrel{\text{def}}{=} \forall \overline{x}.\ \bigwedge_i C_i$, we do a limited number of saturation step, starting from $\{C_i\}_i$. Clauses from the main saturation loop can be used in inferences, emulating a *set-of-support* strategy. If $\bot$ can be derived, the goal is invalid and can be discarded. Many inductive goals, in practice, use computable (recursive) functions. Testing tools such as (Lazy) SmallCheck [32] and QuickCheck [33] are popular options for these properties; in our case, we use *narrowing* [34] because it is readily adapted to rewriting-based functions: the resulting rules are listed in Figure 3. In addition, not all goals contain only computable functions — some functions or predicates might only occur in axioms, not definitions — so we also need to perform the usual superposition inferences.

This mechanism for ruling out invalid goals works quite well in practice, even with a relatively small number of saturation steps. The limit on saturation steps

is a trade-off between the usefulness of detecting invalid conjectures, and the time spent on each candidate lemma.

$$\frac{C \vee s \circ t \qquad l \rightsquigarrow r}{(C \vee s[r]_p)\sigma}\ \textsf{Narrowing} \qquad \frac{C \vee t \qquad l \rightsquigarrow \bigwedge_j D_j}{\bigwedge_j (C \vee D_j)\sigma}\ \textsf{Lit Narrowing}$$

if $l\sigma = s|_p\sigma$, $\circ \in \{\simeq, \not\simeq\}$, $l \rightsquigarrow r$ is a rewrite rule

if $l\sigma = t\sigma$, $t : \textsf{bool}$, $l \rightsquigarrow \bigwedge_j D_j$ is a rewrite rule

Fig. 3: Inference Rules Used for Testing

8 Implementation and Experiments

To evaluate our approach, we implemented it in a superposition prover, Zipperposition [35, chapter 3]. The prover is implemented in OCaml, available at `https://github.com/c-cube/zipperposition` under a permissive BSD license. Thanks to its modular architecture, many extensions of superposition have been added to it, including (as of version 1.2) integer linear arithmetic [35, chapter 4], first-class boolean terms [36], rewrite rules (used for evaluating recursive functions), a simpler version of AVATAR [1], basic support for AC symbols [4], and the inductive reasoning described in this paper. This allows the prover to solve such problems as $\forall (p : \alpha \to \textsf{bool})\ (l : \textsf{list}(\alpha)).\ \textsf{length}(l) \geq \textsf{length}(\textsf{filter}(p, l))$ by using a combination of arithmetic, booleans, and induction. The prover can parse its own native format, TPTP [37], and TIP [17].

In Figure 4, we compare Zipperposition with two variations of CVC4[3] on TIP benchmarks.[4] The first one, CVC4, corresponds to `cvc4 --lang smt --quant-ind` to perform goal-directed induction [12]; the second one, CVC4-gen, has the additional flag `--conjecture-gen` to generate lemmas by theory exploration, like Hipspec [31]. We use `https://github.com/tip-org/tools/` to convert TIP problems into SMT-LIB by removing pattern matching (which is not supported by CVC4). The classical set of IsaPlanner benchmarks [38] are included as a subset of TIP benchmarks. Solvers are given 30 s for each problem, which is a reasonable amount of time a user might wait for automatic provers in a proof assistant. The results are encouraging, since Zipperposition relies on quite simple generalization techniques.

However, our initial aim was to extend superposition provers to do induction, while retaining their efficiency in first-order reasoning. Figure 5 shows a comparison of Zipperposition with some other provers, on TPTP 6.1, which con-

[3] CVC4 1.5-prerelease r6317, see `http://cvc4.cs.stanford.edu/web/`

[4] commit 187b71af8d920d0634b2b8b34c4ac4834b2f6a94 at `https://github.com/tip-org/benchmarks`.

	unsat (/86)	time (s)
Zipperposition	64	4.2
CVC4	67	1.6
CVC4-gen	73	12.4

(a) Results on the IsaPlanner problems

	unsat (/484)	time (s)
Zipperposition	139	53.2
CVC4	138	8.2
CVC4-gen	160	27.7

(b) Results for TIP benchmarks

Fig. 4: Experiments on TIP benchmarks

tains 15,853 first-order problems.[5] CVC4, again, performs quite well; Prover9 is included as a base reference, and E (version 1.9) is one of the best first-order provers. This benchmark shows that Zipperposition keeps good performance on first-order problems.

It is interesting to note that CVC4 is very versatile, and can play on many boards, including first-order logic and inductive theorem proving, in addition to more traditional SMT abilities such as ground reasoning with theories such as arithmetic. We hope that superposition provers will also extend their domain of competency to tackle more expressive logics. Having diverse techniques for automated theorem proving means that portfolio approaches will work well, and will benefit from complementary strengths of the solvers.

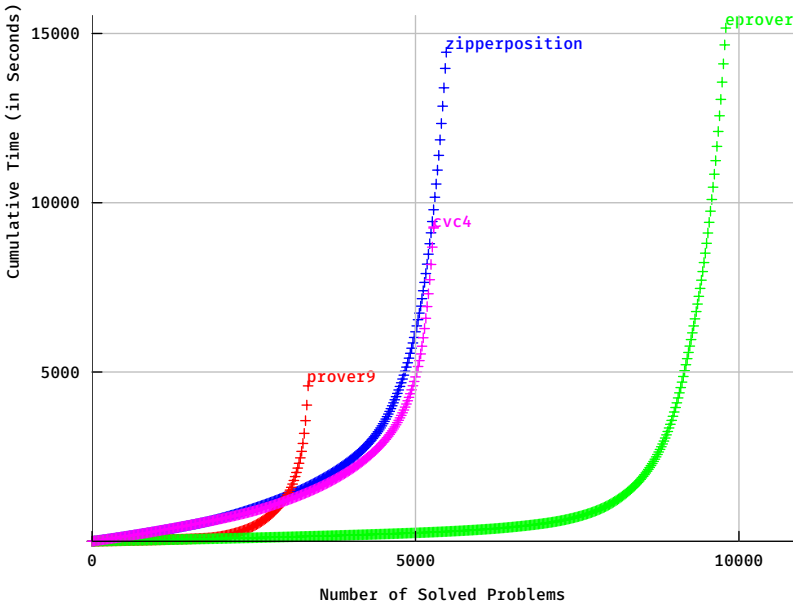
9 Conclusion

In this work, we show a practical integration of inductive reasoning into a superposition prover. Such a combination is desirable because many problems fall outside of either fragment: they might not be purely equational (or Horn), as usually assumed by inductive provers, and yet they might require some inductive reasoning. We introduce a simple inference rule for adding multiple cuts during proof search. Inductive proofs are interleaved with the normal first-order proof search, thanks to AVATAR and this new inference rule.

We also present some techniques for generating inductive subgoals during the proof search, based on Aubin's work [21]; other generalization heuristics, including the ones that rely on theory exploration [31], are compatible with our approach. User-provided lemmas can also be tried and used during proof search, without compromising soundness if they are actually invalid.

Our approach should be relatively straightforward to port to existing state of the art superposition provers. The unsophisticated implementation we describe performs reasonably well on the IsaPlanner and TIP benchmarks. With the addition of more powerful generalization techniques, this suggests that superposition-based first-order provers can become competitive with existing induction provers. In particular, Vampire already supports inductive datatypes and AVATAR split-

[5] experiments on TPTP were run on a 2.20 GHz Intel Xeon® CPU with 30 s timeout and a memory limit of 2 GB.



	solved	unsat	sat	time (s)
E	9802	8840	962	15,160
Zipperposition	5477	4865	612	14,445
CVC4	5282	5253	29	9283
prover9	3341	3341	0	4590

Fig. 5: Results on the first-order fragment of TPTP

ting, and performed very well in SMT-COMP 2016, suggesting superposition is ready to be applied outside of pure first-order logic.

Acknowledgments The author would like to thank Jasmin Blanchette, Gilles Dowek, Guillaume Burel, Pascal Fontaine, and reviewers of previous versions of this paper (one of them, in particular, for pointing out a lot of related works and limitations in several occasions).

References

1. A. Voronkov, "AVATAR: the architecture for first-order theorem provers," in *CAV 2014*, pp. 696–710, 2014.
2. L. C. Paulson and J. C. Blanchette, "Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers," in *IWIL 2010* (G. Sutcliffe, S. Schulz, and E. Ternovska, eds.), EasyChair, 2012.

3. Kaliszyk, Cezary and Urban, Josef, "Learning-assisted automated reasoning with Flyspeck," *Journal of Automated Reasoning*, vol. 53, no. 2, 2014.
4. S. Schulz, "E - a brainiac theorem prover," *AI Commun.*, vol. 15, 2002.
5. A. Riazanov and A. Voronkov, "Vampire 1.1 (system description)," in *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, (London, UK, UK), pp. 376–380, Springer-Verlag, 2001.
6. C. Weidenbach, R. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic, "System Description: SPASS Version 3.0," in *Automated Deduction – CADE-21* (F. Pfenning, ed.), vol. 4603 of *Lecture Notes in Computer Science*, pp. 514–520, Springer, 2007.
7. L. Bachmair and H. Ganzinger, "On Restrictions of Ordered Paramodulation with Simplification," in *10th International Conference on Automated Deduction* (M. E. Stickel, ed.), vol. 449 of *Lecture Notes in Computer Science*, pp. 427–441, Springer, 1990.
8. L. Kovács, S. Robillard, and A. Voronkov, "Coming to terms with quantified reasoning," in *POPL 2017* (G. Castagna and A. D. Gordon, eds.), pp. 260–270, ACM, 2017.
9. M. Kaufmann and J. S. Moore, "ACL2: An industrial strength version of Nqthm," in *Computer Assurance, 1996. COMPASS'96*, pp. 23–34, IEEE, 1996.
10. S. Biundo, B. Hummel, D. Hutter, and C. Walther, "The Karlsruhe induction theorem proving system," in *International Conference on Automated Deduction*, pp. 672–674, Springer, 1986.
11. S. Stratulat, "A unified view of induction reasoning for first-order logic," in *Turing-100, The Alan Turing Centenary Conference*, 2012.
12. A. Reynolds and V. Kuncak, "Induction for SMT solvers," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2015.
13. A. Kersani and N. Peltier, "Combining superposition and induction: A practical realization," in *Frontiers of Combining Systems* (P. Fontaine, C. Ringeissen, and R. Schmidt, eds.), vol. 8152 of *Lecture Notes in Computer Science*, pp. 7–22, Springer Berlin Heidelberg, 2013.
14. M. Beeson, "Otter-lambda, a Theorem-prover with Untyped Lambda-unification," in *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.
15. D. Wand and C. Weidenbach, "Automatic Induction inside Superposition." (unpublished) `http://people.mpi-inf.mpg.de/~dwand/datasup/d.pdf`, April 2017.
16. A. Riazanov and A. Voronkov, "Splitting Without Backtracking," 2001.
17. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "TIP: Tons of Inductive Problems," in *Conferences on Intelligent Computer Mathematics*, pp. 333–337, Springer, 2015.
18. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill, "Rippling: A heuristic for guiding inductive proofs," *Artificial Intelligence*, vol. 62, no. 2, pp. 185 – 253, 1993.
19. R. S. Boyer and J. S. Moore, *A Computational Logic Handbook: Formerly Notes and Reports in Computer Science and Applied Mathematics*. Elsevier, 2014.
20. D. Kapur and M. Subramaniam, *Lemma discovery in automating induction*, pp. 538–552. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.
21. R. Aubin, "Strategies for mechanizing structural induction.," in *IJCAI*, 1977.
22. G. Dowek, T. Hardin, and C. Kirchner, "Theorem Proving Modulo," *Journal of Automated Reasoning*, 2003.
23. G. Burel, *Embedding Deduction Modulo into a Prover*, pp. 155–169. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

24. C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)." `http://www.SMT-LIB.org`, 2016.
25. P. Baumgartner and U. Waldmann, "Hierarchic superposition with weak abstraction," in *Automated Deduction–CADE-24*, Springer, 2013.
26. A. Reynolds and J. C. Blanchette, "A decision procedure for (co)datatypes in SMT solvers," in *CADE-25* (A. Felty and A. Middeldorp, eds.), vol. 9195 of *LNCS*, pp. 197–213, Springer, 2015.
27. C. Barrett, I. Shikanian, and C. Tinelli, "An abstract decision procedure for satisfiability in the theory of inductive data types," *J. Satisf. Boolean Model. Comput.*, vol. 3, pp. 21–46, 2007.
28. L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, ch. 24, pp. 337–340, Berlin, Heidelberg: Springer, 2008.
29. M. Horbach and C. Weidenbach, "Superposition for fixed domains," *ACM Transactions on Computational Logic (TOCL)*, vol. 11, no. 4, p. 27, 2010.
30. H. Zhang, D. Kapur, and M. S. Krishnamoorthy, "A mechanizable induction principle for equational specifications," in *9th International Conference on Automated Deduction* (E. Lusk and R. Overbeek, eds.), vol. 310 of *Lecture Notes in Computer Science*, pp. 162–181, Springer Berlin Heidelberg, 1988.
31. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Hipspec: Automating inductive proofs of program properties.," in *ATx/WInG@ IJCAR*, 2012.
32. C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values," in *Acm sigplan notices*, vol. 44, pp. 37–48, ACM, 2008.
33. K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.
34. F. Lindblad, "Property directed generation of first-order test data.," in *Trends in Functional Programming*, pp. 105–123, Citeseer, 2007.
35. S. Cruanes, *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. PhD thesis, École polytechnique, Sept. 2015.
36. E. Kotelnikov, L. Kovács, G. Reger, and A. Voronkov, "The Vampire and the FOOL," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pp. 37–48, ACM, 2016.
37. G. Sutcliffe, "The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 337–362, 2009.
38. M. Johansson, L. Dixon, and A. Bundy., "Conjecture Synthesis for Inductive Theories," in *Journal of Automated Reasoning*, 2010.