EasyChair Preprint

№ 2468

Induction with Generalization in Superposition Reasoning

Márton Hajdu, Petra Hozzová, Laura Kovács,
Johannes Schoisswohl and Andrei Voronkov

August 2, 2020

Induction with Generalization in Superposition Reasoning

Márton Hajdú[1], Petra Hozzová[1], Laura Kovács[1,2], Johannes Schoisswohl[1,3], and Andrei Voronkov[3,4]

[1] TU Wien, Austria
[2] Chalmers University of Technology, Sweden
[3] University of Manchester, UK
[4] EasyChair

Abstract. We describe an extension of automating induction in superposition-based reasoning by strengthening inductive properties and generalizing terms over which induction should be applied. We implemented our approach in the first-order theorem prover Vampire and evaluated our work against state-of-the-art reasoners automating induction. We demonstrate the strength of our technique by showing that many interesting mathematical properties of natural numbers and lists can be proved automatically using this extension.

1 Introduction

Automating inductive reasoning opens up new possibilities for generating and proving inductive properties, for example properties with inductive data types [21,4] or inductive invariants in program analysis and verification [13,14]. Recent advances related to automating inductive reasoning, such as first-order reasoning with inductively defined data types [16], the Avatar architecture [26], inductive strengthening of SMT properties [22], structural induction in superposition [10] and general induction rules within saturation [19], make it possible to re-consider the grand challenge of mechanizing mathematical induction [5]. In this paper, we contribute to these advances by *generalizing inductive reasoning within the saturation-based proof search* of first-order theorem provers using the superposition calculus.

It is common in inductive theorem proving, that given a formula/goal F, to try to prove a more general goal instead [5]. This makes no sense in saturation-based theorem proving, which is not based on a goal-subgoal architecture. As we aim to automate and generalize inductive reasoning within saturation-based proof search, *our work follows a different approach than the one used in inductive theorem provers.* Namely, our methodology in Section 4 picks up a formula F (not necessarily the goal) in the search space and *adds to the search space new induction axioms with generalization*, that is, instances of generalized induction schemata, aiming at proving both $\neg F$ and a more general formula than $\neg F$. In Section 3 we give a concrete example motivating our approach, illustrating the

advantage of induction with generalization in saturation-based proof search. We then present our main contributions, as follows:

(1) We introduce a new inference rule for first-order superposition reasoning, called *induction with generalization* (Section 4). Our work extends [19] by proving properties with multiple occurrences of the same induction term and by instantiating induction axioms with logically stronger versions of the property being proved. Our approach is conceptually different from previous attempts to use induction with superposition [15,10,11], as we are not restricted to specific clause splitting algorithms and heuristics used in [10], nor are we limited to induction over term algebras with the subterm ordering in [11]. As a result, we stay within the standard saturation framework and do not have to introduce constraint clauses, additional predicates or change the notion of redundancy as in [11].

(2) We implemented our work in the Vampire theorem prover [17] and compared it to state-of-the-art reasoners automating induction, including Acl2 [5], Cvc4 [2], Imandra [18], Zeno [24] and Zipperposition [10] (Section 5). We also provide a set of handcrafted mathematical problems over natural numbers and lists. We show that induction with generalization in Vampire can solve problems that existing systems, including Vampire without this rule, cannot.

(3) We provide a new digital dataset consisting of over 3,300 inductive benchmarks, for which generalized applications of induction is needed (Section 5). Our dataset is formalized within the SMT-LIB format using data types [3] and available at: `https://github.com/vprover/inductive_benchmarks`

2 Preliminaries

We assume familiarity with multi-sorted first-order logic and saturation-based superposition reasoning. For details, we refer to [17]. Throughout this paper we denote fresh Skolem constants by σ, variables by x, y, z and terms by t, all possibly with indices. We denote the equality predicate by $=$ and consider $=$ as part of the language. Further, we write $t_1 \neq t_2$ for the formula $\neg(t_1 = t_2)$.

Given a set of formulas (including a negated conjecture), superposition-based theorem provers run saturation algorithms on a set of clauses corresponding to the clausal normal form (CNF) of the input set of formulas. We denote literals by L and clauses by C, all possibly with indices. We use $\square$ to denote the empty clause. In [16] we showed how superposition-based provers can be extended with reasoning about the theory for finite term algebras.

We will denote term algebras corresponding to natural numbers by $\mathbb{N}$ and lists of natural numbers by $\mathbb{L}$. We refer to the elements of the signature of the term algebras as *constructors*. We will use the same notations $\mathbb{N}$ and $\mathbb{L}$ for these term algebras extended by additional function and predicate symbols shown in Figure 1.

	Natural numbers $\mathbb{N}$	Natural lists $\mathbb{L}$
Constructors	$0 : \mathbb{N}$ $\quad s : \mathbb{N} \to \mathbb{N}$	$\texttt{nil} : \mathbb{L}$ $\quad \texttt{cons} : \mathbb{N} \times \mathbb{L} \to \mathbb{L}$
Symbols	$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$	$+\!\!+ : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$
	$\leq : \mathbb{N} \times \mathbb{N} \to \text{bool}$	$\texttt{prefix} : \mathbb{L} \times \mathbb{L} \to \text{bool}$
Axioms	$\forall y.(0 + y = y)$	$\forall l.(\texttt{nil} +\!\!+ l = l)$
	$\forall x, y.(s(x) + y = s(x + y))$	$\forall x, l, k.(\texttt{cons}(x, l) +\!\!+ k = \texttt{cons}(x, l +\!\!+ k))$
	$\forall x.0 \leq x$	$\forall l.\texttt{prefix}(\texttt{nil}, l)$
	$\forall x.\neg s(x) \leq 0$	$\forall x, l.\neg\texttt{prefix}(\texttt{cons}(x, l), \texttt{nil})$
	$\forall x, y.\big(s(x) \leq s(y) \leftrightarrow x \leq y\big)$	$\forall x, l, y, k.\big(\texttt{prefix}(\texttt{cons}(x, l), \texttt{cons}(y, k)) \leftrightarrow (x = y \wedge \texttt{prefix}(l, k))\big)$

Fig. 1. Term algebras of $\mathbb{N}$ and $\mathbb{L}$, together with additional symbols and axioms.

Specifically, we will deal with $+$ and $\leq$ for $\mathbb{N}$ having their standard meaning and $+\!\!+$ and $\texttt{prefix}$ for $\mathbb{L}$, denoting the list concatenation and the prefix relation, respectively. These additional symbols are axiomatized by first-order formulas corresponding to their recursive definitions, shown in Figure 1.

While we use $\mathbb{N}$ and $\mathbb{L}$ for illustration, we however note that our approach can be used for proving properties over any other theories with various forms of induction.

Theorem proving of first-order properties of inductively defined data types needs to handle the domain closure, injectivity, distinctness and acyclicity axioms of term algebras – a detailed definition of these axioms can be found in [23,16]. The challenge we address in [16] is how to automate proving term algebras properties given the fact that the acyclicity axiom is not finitely axiomatizable.

Throughout this paper, we will be using the structural induction axiom and rule for $\mathbb{N}$, introduced in [19], for illustrating our approach. Given a literal $\neg L[t]$, where t is chosen as an induction term, a structural induction axiom for $\mathbb{N}$ is:

$$\big(L[0] \wedge \forall x.(L[x] \to L[s(x)])\big) \to \forall y.(L[y]). \tag{1}$$

Informally, the axiom expresses that if the base case holds, and if the induction step holds, then the literal holds for all possible values. The structural induction rule for $\mathbb{N}$, given a clause $\neg L[t] \vee C$, adds a clausified form of this axiom to the search space:

$$\frac{\neg L[t] \vee C}{(\neg L[0] \vee L[\sigma] \vee L[y]) \wedge (\neg L[0] \vee \neg L[s(\sigma)] \vee L[y])}. \tag{2}$$

After using the rule, the $L[y]$ in both resulting clauses can be resolved against the $\neg L[t]$ in the premise clause.

3 Motivating Example

Let us now motivate our approach to induction with generalization, by considering the following formula expressing the associativity of addition over $\mathbb{N}$:

$$\forall x, y, z.(x + (y + z) = (x + y) + z), \qquad \text{with } x, y, z \in \mathbb{N}. \tag{3}$$

(C_1) $\sigma_1 + (\sigma_2 + \sigma_3) \not\approx (\sigma_1 + \sigma_2) + \sigma_3$ [input]
(C_2) $0 + (\sigma_2 + \sigma_3) \not\approx (0 + \sigma_2) + \sigma_3 \vee \sigma + (\sigma_2 + \sigma_3) \approx (\sigma + \sigma_2) + \sigma_3$ [induct. C_1]
(C_3) $0 + (\sigma_2 + \sigma_3) \not\approx (0 + \sigma_2) + \sigma_3 \vee s(\sigma) + (\sigma_2 + \sigma_3) \not\approx (s(\sigma) + \sigma_2) + \sigma_3$ [induct. C_1]
(C_4) $0 + (\sigma_2 + \sigma_3) \not\approx (0 + \sigma_2) + \sigma_3 \vee s(\sigma + (\sigma_2 + \sigma_3)) \not\approx s((\sigma + \sigma_2) + \sigma_3)$ [C_3 + axiom]
(C_5) $0 + (\sigma_2 + \sigma_3) \not\approx (0 + \sigma_2) + \sigma_3 \vee \sigma + (\sigma_2 + \sigma_3) \not\approx (\sigma + \sigma_2) + \sigma_3$ [injective C_4]
(C_6) $0 + (\sigma_2 + \sigma_3) \not\approx (0 + \sigma_2) + \sigma_3$ [res. C_2, C_5]
(C_7) $\sigma_2 + \sigma_3 \not\approx \sigma_2 + \sigma_3$ [C_6 + axiom]
(C_8) $\square$ [trivial ineq. C_7]

Fig. 2. Proof of associativity of + in a saturation-based theorem prover with induction

The induction approach introduced in [19] is able to prove this problem. The main steps of such a proof are shown in Figure 2 and discussed next. First, the negation of formula (3) is skolemized, yielding the (unit) clause C_1 of Figure 2. As already mentioned, the σ_i denote fresh Skolem constants introduced during clausification. Next, the structural induction axiom (1) is instantiated so that its conclusion can resolve against C_1 using the constant σ_1 as the induction term, resulting in the formula:

$$\begin{aligned}&\big(0 + (\sigma_2 + \sigma_3) \approx (0 + \sigma_2) + \sigma_3 \wedge \\ &\ \forall x.(x + (\sigma_2 + \sigma_3) \approx (x + \sigma_2) + \sigma_3 \rightarrow s(x) + (\sigma_2 + \sigma_3) \approx (s(x) + \sigma_2) + \sigma_3)\big) \\ &\rightarrow \forall y.(y + (\sigma_2 + \sigma_3) \approx (y + \sigma_2) + \sigma_3).\end{aligned} \tag{4}$$

Then, the CNF of the induction axiom (4) is added to the search space using the following instance of the structural induction rule (2):

$$\frac{\sigma_1 + (\sigma_2 + \sigma_3) \not\approx (\sigma_1 + \sigma_2) + \sigma_3}{\begin{array}{c}(0 + (\sigma_2 + \sigma_3) \not\approx (0 + \sigma_2) + \sigma_3 \vee \sigma + (\sigma_2 + \sigma_3) \approx (\sigma + \sigma_2) + \sigma_3 \vee \\ y + (\sigma_2 + \sigma_3) \approx (y + \sigma_2) + \sigma_3) \\ \wedge \\ (0 + (\sigma_2 + \sigma_3) \not\approx (0 + \sigma_2) + \sigma_3 \vee s(\sigma) + (\sigma_2 + \sigma_3) \not\approx (s(\sigma) + \sigma_2) + \sigma_3 \vee \\ y + (\sigma_2 + \sigma_3) \approx (y + \sigma_2) + \sigma_3)\end{array}}. \tag{5}$$

The clauses from the inference conclusion are resolved against C_1, yielding clauses C_2, C_3 of Figure 2. Clause C_4 originates by repeated demodulation into C_3 using the second axiom of Figure 1 over $\mathbb{N}$. Further, C_5 is derived from C_4 by using the injectivity property of term algebras and C_6 is a resolvent of C_2 and C_5. Clause C_7 is then derived by repeated demodulation into C_6, using the the first axiom of Figure 1 over $\mathbb{N}$. By removing the trivial inequality from C_7, we finally derive the empty clause C_8.

Consider now the following instance of the associativity property (3):

$$\forall x.(x + (x + x) \approx (x + x) + x). \tag{6}$$

While (6) is an instance of (3), we cannot prove it using the same approach. Let us explain why this is the case. By instantiating the induction axiom (1) using (6), we get:

$$\begin{aligned}&\big(0 + (0 + 0) \approx (0 + 0) + 0 \wedge \\ &\ \forall x.(x + (x + x) \approx (x + x) + x \rightarrow s(x) + (s(x) + s(x)) \approx (s(x) + s(x)) + s(x))\big) \\ &\rightarrow \forall y.(y + (y + y) \approx (y + y) + y).\end{aligned} \tag{7}$$

After resolving this axiom with the skolemized negation of (6), we get the following two clauses[5]:

$$0 + (0 + 0) \not\approx (0 + 0) + 0 \vee \sigma + (\sigma + \sigma) = (\sigma + \sigma) + \sigma \tag{8}$$

$$0 + (0 + 0) \not\approx (0 + 0) + 0 \vee s(\sigma) + (s(\sigma) + s(\sigma)) \not\approx (s(\sigma) + s(\sigma)) + s(\sigma) \tag{9}$$

While the first literals of (8) and (9) are easily resolved using axioms of $+$, not much can be done with the latter literals. We can only apply repeated demodulations over the second literal of (9) using axioms of $+$ and the injectivity property of term algebras, yielding $\sigma + s(\sigma + s(\sigma)) \not\approx (\sigma + s(\sigma)) + s(\sigma)$. No further inference over this formula can be applied, in particular it cannot be resolved against the second literal of (8). Hence, the approach of [19] fails proving (6).

The existing approaches to induction also suffer from the same problem. For example [5,18,2,24,10], can prove property (3) but fail to prove its weaker instance (6). The common recipe in inductive theorem proving [5] is to try to prove (3) in addition to trying to prove (6).

Interestingly, in saturation-based theorem proving we can do better. If we follow the common recipe, we would add a generalized goal and then an induction axiom for it. Instead, we only add the induction axiom instance corresponding to the generalized goal without adding the extra goal, which results in a smaller number of clauses. More precisely, in addition to the instance of the induction schema corresponding to (6), we also add instance (4) corresponding to (3). We call this new inference rule *induction with generalization*.

4 Induction with Generalization

Following [19], we consider an *induction axiom* to be any valid formula of the form $premise \rightarrow \forall y.(L[y])$, in the underlying theory, such as the theory of term algebras. An example of an induction axiom is the structural induction axiom (4). An *induction schema* is a collection of induction axioms. Each induction schema we consider is the set of first-order instances of some valid higher-order formula. The work [19] introduces a rule of induction where a ground literal $\neg L[t]$ appearing in the proof search triggers addition of the corresponding induction axioms $premise \rightarrow \forall y.(L[y])$ to the search space:

$$\frac{\neg L[t] \vee C}{\mathrm{CNF}(premise \rightarrow \forall y.(L[y]))}\ (\texttt{induction}), \tag{10}$$

where $L[y]$ is obtained from $L[t]$ by replacing *all* occurrences of t by y. An example of an instance of the induction rule is (5).

While addition of a large number of such formulas may seem to blow up the search space, in practice Vampire handles such addition with little overhead, resulting in finding proofs containing nearly 150 induction inferences [19]. The reason why the overhead of adding structural induction axioms is small is explained in [20]: the added clauses only contain one variable (the y in $L[y]$), and

[5] These clauses are instances of C_2 and C_3 from Figure 2.

the clauses containing this literal are immediately subsumed by a ground clause. The net result is adding a small number of ground clauses, which are especially easy to handle in the Avatar architecture implemented in Vampire.

Induction with generalization. In a nutshell, given a goal, we add an induction axiom corresponding to a more general one. The rule can be formulated in the same way as (10), yet with a different conclusion:

$$\frac{\neg L[t] \vee C}{\mathrm{CNF}(\mathit{premise}' \to \forall y.(L'[y]))}\ (\texttt{IndGen}), \tag{11}$$

where $L'[y]$ is obtained from $L[t]$ by replacing *some* occurrences of t by y, and $\mathit{premise}'$ is the premise corresponding to $L'[y]$. Both induction rules are obviously sound because their conclusions are constructed such that they are valid in the underlying theory.

To implement `IndGen`, if a clause selected for inferences contains a ground literal $\neg L[t]$ having more than one occurrence of t, we should select a non-empty subset of occurrences of t in $L[t]$, select an induction axiom corresponding to this subset, and then apply the rule.

Motivating example, continued. Suppose that t is σ_1 and $\neg L[t]$ is $\sigma_1 + (\sigma_1 + \sigma_1) \not\approx (\sigma_1 + \sigma_1) + \sigma_1$, which is obtained by negating and skolemizing (6). Then by applying `IndGen` we can add the following induction axiom:

$$\begin{aligned}&\big(0 + (\sigma_1 + \sigma_1) = (0 + \sigma_1) + \sigma_1 \wedge \\ &\ \forall x.(x + (\sigma_1 + \sigma_1) = (x + \sigma_1) + \sigma_1 \to s(x) + (\sigma_1 + \sigma_1) = (s(x) + \sigma_1) + \sigma_1)\big) \\ &\to \forall y.(y + (\sigma_1 + \sigma_1) = (y + \sigma_1) + \sigma_1),\end{aligned} \tag{12}$$

which is different from (7). When we add this formula, we can derive the empty clause in the same way as in Figure 2.

Saturation with induction with generalization. The main questions to answer when applying induction with generalization is which occurrences of the induction term in the induction literal we should choose.

Generally, if the subterm t occurs n times in the premise, there are $2^n - 1$ ways of applying the rule, all potentially resulting in *formulas not implying each other*. Thus, an obvious heuristic to use *all* non-empty subsets may result in too many formulas. For example, $\sigma_1 + (\sigma_1 + \sigma_1) \not\approx (\sigma_1 + \sigma_1) + \sigma_1$ would result in adding 63 induction formulas.

Another simple heuristic is to restrict the number of occurrences selected as induction term to a fixed number. This strategy reduces the number of applications of induction at the cost of losing proofs that would need subsets of cardinality larger than the limit. Finding possible heuristics for selecting specific subsets for common cases of literals can be subject of future work, especially interesting in proof assistants in mathematics.

Note that some of the conclusions of (11) can, in turn, have many children obtained by induction with generalization. Our experiments in Section 5 show that, even when we generate all possible children, Vampire can still solve large

examples with more than 10 occurrences of the same induction variable, again thanks to the effect that, for each application of induction, only a small number of ground clauses turn out to be added to the search space.

We therefore believe that our work can potentially be also useful for larger examples, and even in cases when the inductive property to be proved is embedded in a larger context.

5 Experiments

Implementation. We implemented induction with generalization in Vampire, with two new options:

- boolean-valued option `indgen`, which turns on/off the application of induction with generalization, with the default value being off, and
- integer-valued option `indgenss`, which sets the maximum size of the subset of occurrences used for induction, with the default value 3. This option is ignored if `indgen` is off.

Our implementation of induction with generalization is available at: `https://github.com/vprover/vampire`.
In experiments described here, if `indgen` is off, Vampire performs induction on all occurrences of a term in a literal as in [19]. In this section

- Vampire refers to the (default) version of Vampire with induction rule (10) (i.e., the option `-ind struct`)
- Vampire* additionally uses the `IndGen` rule of induction with generalization (11) (i.e., the options `-ind struct -indgen on`).
- Vampire** uses the same options as Vampire* plus the option `-indoct on`, which applies induction to arbitrary ground terms, not just to constants as in Vampire or in Vampire*.

SMT-LIB experiments. We evaluated our work using the UFDT and UFDTLIA problem sets from SMT-LIB [3], yielding all together 4854 problems. Many of these problems come from program analysis and verification and contain large numbers of axioms, so they are different from standard mathematical examples used in many other papers on automation of induction. Given the nature of the benchmarks, we were interested in two questions:

1. What is the overhead incurred by using induction with generalization in large search spaces, especially when it is not used in proofs? If the new rule is prohibitively expensive, this means it could probably only be used in smaller examples used in interactive theorem proving.
2. Is the new rule useful at all for for this kind of benchmarks? While the new rule can be used in principle, should it (or can it) be used in program analysis and verification?

Our results show that *the overhead is relatively small* but we could not solve problems not solvable without the use of the new rule.

Induction (10) in Vampire was already evaluated in [19] against other solvers on these examples. Hence, we only compare how Vampire*/Vampire** performs against Vampire, using both the default and the portfolio modes. (In the default mode, Vampire/Vampire*/Vampire** uses default values for all parameters except the ones specified by the user; in the portfolio mode, Vampire/Vampire*/Vampire** sequentially tries different configurations for parameters not specified by the user.) Together, we ran 18 instances: Vampire, Vampire* with `indgenss` set to 2, 3, 4 and unlimited, and Vampire** with the same four variants of `indgenss`; each of them in both default and portfolio mode. We ran our experiments on the StarExec cluster [25].

The best Vampire*/Vampire** solved 5 problems in the portfolio mode and 1 problem in the default mode not solved by Vampire. However, the proofs found by them did not use induction with generalization. This is a common problem in experiments with saturation theorem proving: new rules change the direction of the proof search and may result in new simplifications that also drastically affect the search space. As a result, new proofs may be found, yet these proofs do not actually use the new rule. There were no problems solved by Vampire that were not solved by any Vampire*/Vampire**.

The maximum number of `IndGen` applications in proofs was 3 and the maximum depth of induction was 4. Vampire*/Vampire** used generalized induction in proofs of 10 problems. However, these problems are also solvable by Vampire (without generalized induction). Thus, we conclude that SMT-LIB problems (probably as well as other typical program analysis and verification benchmarks) typically do not gain from using generalization.

Experiments with mathematical problems. We handcrafted a number of natural problems over natural numbers and lists and tested the new rule on these problems. Our benchmarks are available at: `https://github.com/vprover/inductive_benchmarks`.

Table 1 lists 16 of such examples using the functions defined in Figure 1. Some examples were taken from or inspired by the TIP benchmark library [9]: e.g., the seventh benchmark in Table 1 is adapted from the TIP library and the second problem is inspired by a symmetric problem from the TIP library, $\forall x.(s(x)+x = s(x+x))$. While they are handcrafted, we believe they are representative since no attempt was done to exclude problems not solvable by Vampire using induction with generalization.

We evaluated and compared several state-of-the-art reasoners supporting standard input formats and, due to the nature of our work, either superposition-based approaches or approaches to generalization. It was not easy to make these experiments since provers use different input syntaxes (see Table 2). As a result, we also had to design translations of our benchmarks.

Except for Imandra (which is a cloud-based service), we ran our experiments on a 2,9 GHz Quad-Core Intel Core i7 machine. We ran each solver as a single-threaded process with a 5 second time limit. Our results are summa-

Table 1. Experiments with 16 handcrafted benchmarks. "✓" denotes success, "–" denotes failure.

	Theory	Vampire*	Vampire**	Vampire	Cvc4	Zipperposition	Zeno	Imandra	Acl2	Cvc4-Gen	ZipRewrite
$\forall x, y.(x + y = y + x)$	$\mathbb{N}$	✓	✓	✓	✓	✓	✓	✓	–	✓	✓
$\forall x.(x + s(x) = s(x + x))$		✓	✓	–	–	–	–	–	–	✓	✓
$\forall x, y, z.(x + (y + z) = (x + y) + z)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x.(x + (x + x) = (x + x) + x)$		✓	✓	–	–	–	✓	–	–	✓	✓
$\forall x.((x+x)+((x+x)+x) = x+(x+((x+x)+x)))$		✓	✓	–	–	–	✓	–	–	✓	✓
$\forall x, y.(y + (x + x) = (x + y) + x)$		✓	✓	–	–	–	–	–	–	–	✓
$\forall x.(x \leq x)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x, y.(x \leq x + y)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x.(x \leq x + x)$		✓	✓	–	–	–	–	–	–	–	–
$\forall x.(x + x \leq (x + x) + x)$		✓	✓	–	–	–	✓	–	–	–	–
$\forall l, k, j.(l +\!\!+ (k +\!\!+ j) = (l +\!\!+ k) +\!\!+ j)$	$\mathbb{L}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall l.(l +\!\!+ (l +\!\!+ l) = (l +\!\!+ l) +\!\!+ l)$		✓	✓	–	–	–	–	–	–	–	✓
$\forall l, k.(l +\!\!+ (k +\!\!+ (l +\!\!+ l)) = (l +\!\!+ k) +\!\!+ (l +\!\!+ l))$		✓	✓	–	–	–	–	–	–	–	✓
$\forall l, k.\texttt{prefix}(l, l +\!\!+ k)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall l.\texttt{prefix}(l, l +\!\!+ l)$		✓	✓	–	–	–	–	–	–	–	–
$\forall l : \mathbb{L}, x : \mathbb{N}.(\texttt{cons}(x + s(x), l) +\!\!+ (l +\!\!+ l) = (\texttt{cons}(s(x) + x, l) +\!\!+ l) +\!\!+ l)$	$\mathbb{N}, \mathbb{L}$	✓	✓	–	–	–	–	–	–	–	–

rized in Table 1, where Cvc4-Gen refers to the solver Cvc4 with the automatic lemma discovery enabled. ZipRewrite refers to Zipperposition with function and predicate definitions encoded as rewrite rules instead of ordinary logical formulas, in order to trigger its generalization heuristics [10]. Configurations used for running all solvers are listed in Table 2.

Table 1 shows that Vampire*/Vampire** (with `indgenss=3`) outperforms all solvers, including Vampire itself. When considering solvers without fine-tuned heuristics, such as in ZipRewrite and Cvc4-Gen, Vampire** solves many more problems. Interestingly, ZipRewrite heuristics work well with addition and list concatenation, but not with orders. Further, Cvc4-Gen heuristics prove associativity of addition, but not the list counterpart for concatenation. We believe our experiments show the potential of using induction with generalization as a new inference rule since it outperforms heuristic-driven approaches with no special heuristics or fine-tuning added to Vampire.

Experiments with problems requiring associativity and commutativity. The $(x + x) + x = x + (x + x)$ is a special case of a family of problems over natural numbers. The problems can be formulated as follows.

Let t_1 and t_2 be two terms built using variables, $+$ and the successor function. Then the equality $t_1 = t_2$ is valid over natural numbers if and only if they have the same number of occurrences of the successor function and each variable of this equality has the same number of occurrences in t_1 and t_2. For example, the following equality is valid over natural numbers:

Table 2. Configurations and input format of solvers for the mathematical problems.

Solver	Configuration	Input format
Vampire	`-ind struct`	SMT-LIB
Vampire*	`-ind struct -indgen on`	SMT-LIB
Vampire**	`-ind struct -indgen on -indoct on`	SMT-LIB
Cvc4	`--quant-ind`	SMT-LIB
Cvc4-Gen	`--quant-ind --conjecture-gen`	SMT-LIB
Zipperposition	default mode	`.zf` (native input format)
ZipRewrite	default mode	`.zf` with definitions as rewrite rules
Zeno	default mode	functional program encoding
Imandra	default mode	functional program encoding
Acl2	default mode	functional program encoding

$$s(x + (x + s(y + z))) + s(z) = (z + s(x)) + (x + s(s((z + y)))).$$

To prove such problems over $\mathbb{N}$, one needs both induction and generalization. Without the successor function, they can be easily proved using associativity and commutativity of $+$, but associativity and commutativity are not included in the axioms of $\mathbb{N}$. When the terms are large, the problems become highly challenging.

We generated a set of instances of these problems (with and without the successor function, and also other functions and predicates) by increasing term sizes. We also generated similar problems for lists using concatenation and reverse functions, and prefix predicate. Some of the terms were, e.g., variations of (6) with 20 occurrences of x. Our entire dataset, containing over 3,300 examples, is available at the previously mentioned URL.

We were again interested in evaluating and comparing various reasoners and approaches on these problems. The interesting feature of these problems is that they are natural yet we can generate problems of almost arbitrary complexity.

We evaluated and compared Vampire*, Vampire**, Cvc4-Gen, Zeno and ZipRewrite, that is the best performing solvers on inductive reasoning with generalization according to Table 1, using the same experimental setting as already described for Table 1. Table 3 lists a partial summary of our experiments, displaying results for 2,007 large instances of four simple properties with one variable, corresponding to the fourth, ninth, twelfth and fifteenth problem from Table 1. (Due to space constraints, we chose these problems as a representative subset of our large benchmarks, since the solvers' performance was very similar for the whole benchmark set.)

In Table 3, we use the following notation. By $nx = nx$ we denote formulas of the form $x \circ ... \circ x = x \circ ... \circ x$ with n occurrences of x on both sides of the equality, and parentheses on various places in the expressions, with $\circ$ being $+$, or $++$ for the datatypes $\mathbb{N}$ and $\mathbb{L}$, respectively. By $mx \leq nx$ and $\texttt{prefix}(mx, nx)$ we

denote formulas of the form $x+...+x \leq x+...+x$ and $\texttt{prefix}(x \mathbin{+\!\!+} ... \mathbin{+\!\!+} x, x \mathbin{+\!\!+} ... \mathbin{+\!\!+} x)$, respectively, with m occurrences of x on the left and n occurrences of x on the right hand side of the $\leq$ or `prefix` predicates, and with parentheses on various places in the expressions. Result $N\%(M)$ means that the solver solved M of the problems from this category, which corresponds to $N\%$.

From Table 3, we conclude that Vampire** scales better than Cvc4-Gen on a large majority of benchmarks, and scales comparably to Zeno. While ZipRewrite can solve more problems than Vampire**, Vampire** is more consistent in solving at least some problems from each category. ZipRewrite can solve many problems thanks to treatment of equalities as rewrite rules. We are planning to add an option of using recursive definitions as rewrite rules in Vampire in the future too.

6 Related Work

Research into automating induction has a long history with a number of techniques developed, including for example approaches based on semi-automatic inductive theorem proving [5,7,18,8], specialized rewriting procedures [12], SMT reasoning [22] and superposition reasoning [15,10,19,11].

Previous works on automating induction mainly focus on inductive theorem proving [7,8,24]: deciding when induction should be applied and what induction axiom should be used. Further restrictions are made on the logical expressiveness, for example induction only over universal properties [5,24] and without uninterpreted symbols [18], or only over term algebras [15,11]. Inductive proofs usually rely on auxiliary lemmas to help proving an inductive property. In [8] heuristics for finding such lemmas are introduced, for example by randomly generating equational formulas over random inputs and using these formulas if they hold reasonably often. The use of [8] is therefore limited to the underlining heuristics. Other approaches to automating induction circumvent the need for auxiliary lemmas by using uncommon cut-free proof systems for inductive reasoning, such as a restricted ω-rule [1], or cyclic reasoning [6].

The work presented in this paper automates induction by integrating it directly in superposition-based proof search, without relying on rewrite rules and external heuristics for generating auxiliary inductive lemmas/subgoals as in [7,8,24,5,18]. Our new inference rule `IndGen` for induction with generalization adds new formulas to the search space and can replace lemma discovery heuristics used in [7,8,22]. Our work also extends [19] by using and instantiating induction axioms with logically stronger versions of the property being proved. Unlike [10], our methods do not necessarily depend on Avatar [26], can be used with any (inductive) data type and target induction rules different than structural induction. Contrarily to [11], we are not limited to induction over term algebras with the subterm ordering and we stay in a standard saturation framework. Moreover, compared to [5,7,8,22], one of the main advantages of our approach is that it does not use a goal-subgoal architecture and can, as a result, combine superposition-based equational reasoning with inductive reasoning.

Table 3. Experiments on 2,007 arithmetical problems.

	Theory	Vampire*	Vampire**	Cvc4-Gen	Zeno	ZipRewrite
$3x = 3x$	$\mathbb{N}$	100% (1)	100% (1)	100% (1)	100% (1)	100% (1)
$4x = 4x$		90% (9)	100% (10)	100% (10)	20% (2)	100% (10)
$5x = 5x$		30% (15)	50% (25)	100% (50)	12% (6)	100% (50)
$6x = 6x$		8% (4)	18% (9)	100% (50)	22% (11)	100% (50)
$7x = 7x$		–	10% (5)	100% (50)	2% (1)	100% (50)
$8x = 8x$		–	2% (1)	100% (50)	4% (2)	100% (50)
$9x = 9x$		–	2% (1)	100% (50)	8% (4)	84% (42)
$10x = 10x$		–	–	100% (50)	8% (4)	90% (45)
$3x = 3x$	$\mathbb{L}$	100% (1)	100% (1)	–	–	100% (1)
$4x = 4x$		70% (7)	90% (9)	–	–	100% (10)
$5x = 5x$		46% (23)	48% (24)	–	–	100% (50)
$6x = 6x$		6% (3)	26% (13)	–	6% (3)	100% (50)
$7x = 7x$		2% (1)	6% (3)	–	–	100% (50)
$8x = 8x$		–	–	–	–	90% (45)
$9x = 9x$		–	–	–	–	88% (44)
$10x = 10x$		–	–	–	–	68% (34)
$3x \leq 3x$	$\mathbb{N}$	100% (2)	100% (2)	100% (2)	100% (2)	100% (2)
$4x \leq 4x$		–	15% (3)	100% (20)	20% (4)	100% (20)
$5x \leq 5x$		–	4% (2)	100% (50)	12% (6)	100% (50)
$1x \leq 2x$		100% (1)	100% (1)	–	–	–
$2x \leq 3x$		50% (1)	50% (1)	–	100% (2)	–
$3x \leq 4x$		–	30% (3)	–	40% (4)	–
$4x \leq 5x$		–	8% (4)	–	16% (8)	–
$5x \leq 6x$		–	6% (3)	–	10% (5)	–
$1x \leq 3x$		100% (2)	100% (2)	–	100% (2)	100% (2)
$2x \leq 4x$		–	40% (2)	–	40% (2)	100% (5)
$3x \leq 5x$		–	14% (4)	–	28% (8)	100% (28)
$4x \leq 6x$		–	10% (5)	–	18% (9)	100% (50)
$5x \leq 7x$		–	4% (2)	–	18% (9)	100% (50)
$1x \leq 4x$		100% (5)	100% (5)	–	80% (4)	100% (5)
$2x \leq 5x$		–	35% (5)	–	42% (6)	100% (14)
$3x \leq 6x$		–	18% (9)	–	38% (19)	100% (50)
$4x \leq 7x$		–	6% (3)	–	16% (8)	100% (50)
$5x \leq 8x$		–	–	–	6% (3)	100% (50)
$1x \leq 5x$		100% (14)	100% (14)	–	85% (12)	100% (14)
$2x \leq 6x$		–	33% (14)	–	26% (11)	100% (42)
$3x \leq 7x$		–	14% (7)	–	32% (16)	100% (50)
$4x \leq 8x$		–	4% (2)	–	18% (9)	100% (50)
$5x \leq 9x$		–	–	–	14% (7)	100% (50)
$\texttt{prefix}(3x, 3x)$	$\mathbb{L}$	100% (2)	50% (1)	–	–	100% (2)
$\texttt{prefix}(4x, 4x)$		–	25% (5)	–	–	100% (20)
$\texttt{prefix}(5x, 5x)$		–	2% (1)	–	4% (2)	100% (50)
$\texttt{prefix}(1x, 2x)$		100% (1)	100% (1)	–	–	–
$\texttt{prefix}(2x, 3x)$		–	50% (1)	–	50% (1)	–
$\texttt{prefix}(3x, 4x)$		–	20% (2)	–	20% (2)	–
$\texttt{prefix}(4x, 5x)$		–	8% (4)	–	8% (4)	–
$\texttt{prefix}(5x, 6x)$		–	–	–	–	–
$\texttt{prefix}(1x, 3x)$		100% (2)	100% (2)	–	50% (1)	100% (2)
$\texttt{prefix}(2x, 4x)$		20% (1)	40% (2)	–	20% (1)	100% (5)
$\texttt{prefix}(3x, 5x)$		–	14% (4)	–	14% (4)	100% (28)
$\texttt{prefix}(4x, 6x)$		–	6% (3)	–	8% (4)	100% (50)
$\texttt{prefix}(5x, 7x)$		–	2% (1)	–	2% (1)	100% (50)
$\texttt{prefix}(1x, 4x)$		100% (5)	100% (5)	–	40% (2)	100% (5)
$\texttt{prefix}(2x, 5x)$		–	35% (5)	–	21% (3)	100% (14)
$\texttt{prefix}(3x, 6x)$		–	14% (7)	–	12% (6)	100% (50)
$\texttt{prefix}(4x, 7x)$		–	4% (2)	–	4% (2)	100% (50)
$\texttt{prefix}(5x, 8x)$		–	–	–	4% (2)	100% (50)
$\texttt{prefix}(1x, 5x)$		100% (14)	100% (14)	–	42% (6)	100% (14)
$\texttt{prefix}(2x, 6x)$		–	33% (14)	–	21% (9)	100% (42)
$\texttt{prefix}(3x, 7x)$		–	16% (8)	–	16% (8)	100% (50)
$\texttt{prefix}(4x, 8x)$		–	10% (5)	–	12% (6)	100% (50)
$\texttt{prefix}(5x, 9x)$		–	–	–	–	100% (50)

Normally, generalization in theorem proving means that given a goal F, we try to prove a more general goal. In logic, a statement F' is more general than F if F' implies F. Thus, by proving F' we also prove F. One way to generalize is to replace one or more occurrences of a subterm by a fresh variable, using the fact that $\forall x.(F[x])$ implies $F[t]$. This is essentially the idea behind approaches to generalization in all systems we compared with. While our approach is superficially similar, it does something *fundamentally different*. Instead of (or in addition to) adding an instance I of the induction schema that can be used to prove $F[t]$, we add an instance I' that can be used to prove $\forall x.(F[x])$. An interesting observation is that, in general, neither I implies I', nor I' implies I, so neither of I and I' is more general.

The *second fundamental difference* is that, because induction in Vampire is not based on a goal-subgoal architecture, we can add both induction formulas I and I' at the same time. While this may seem inefficient, for some induction schemata, including structural induction, the overhead is very small (as also confirmed by our experiments).

7 Conclusions

We introduced a new rule for induction with generalization in saturation-based reasoning based on adding induction axioms for proving generalizations of the goals appearing during proof-search. Our experiments show that we solve many problems that other existing systems cannot solve. Future work includes designing heuristics to guide proof search, using rewriting approaches, and performing other kinds of generalization and induction.

Acknowledgements

We thank Giles Reger for discussions related to the work. We acknowledge funding supporting this work, in particular the ERC starting grant 2014 SYMCAR 639270, the EPSRC grant EP/P03408X/1, the ERC proof of concept grant 2018 SYMELS 842066, the Wallenberg Academy fellowship 2014 TheProSE, the Austrian FWF research project W1255-N23, and the Hungarian-Austrian project 101öu8.

References

1. Baker, S., Ireland, A., Smaill, A.: On the Use of the Constructive Omega-Rule within Automated Deduction. In: Proc. of LPAR. pp. 214–225 (1992)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. of CAV. pp. 171–177. Springer (2011)
3. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)

4. Blanchette, J.C., Peltier, N., Robillard, S.: Superposition with Datatypes and Codatatypes. In: Proc. of IJCAR. pp. 370–387 (2018)
5. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook, Perspectives in computing, vol. 23. Academic Press (1979)
6. Brotherston, J., Simpson, A.: Sequent calculi for induction and infinite descent. J. Log. Comput. **21**(6), 1177–1216 (2011)
7. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., Smaill, A.: Rippling: A Heuristic for Guiding Inductive Proofs. Artif. Intell. **62**(2), 185–253 (1993)
8. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: HipSpec: Automating Inductive Proofs of Program Properties. In: Proc. of ATx/WInG. pp. 16–25 (2012)
9. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: TIP: Tons of Inductive Problems. In: Proc. of CICM. pp. 333–337. Springer (2015)
10. Cruanes, S.: Superposition with Structural Induction. In: Proc. of FRoCoS. pp. 172–188 (2017)
11. Echenheim, M., Peltier, N.: Combining Induction and Saturation-Based Theorem Proving. J. Automated Reasoning **64**, 253–294 (2020)
12. Falke, S., Kapur, D.: Rewriting Induction + Linear Arithmetic = Decision Procedure. In: Proc. of IJCAR. pp. 241–255 (2012)
13. Gleiss, B., Kovács, L., Robillard, S.: Loop Analysis by Quantification over Iterations. In: Proc. of LPAR. pp. 381–399 (2018)
14. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on Demand. In: Prof. of ATVA. pp. 248–266 (2018)
15. Kersani, A., Peltier, N.: Combining Superposition and Induction: A Practical Realization. In: Proc. of FroCoS. pp. 7–22 (2013)
16. Kovács, L., Robillard, S., Voronkov, A.: Coming to Terms with Quantified Reasoning. In: Proc. of POPL. pp. 260–270 (2017)
17. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: Proc. of CAV. pp. 1–35 (2013)
18. Passmore, G., Cruanes, S., Ignatovich, D., Aitken, D., Bray, M., Kagan, E., Kanishev, K., Maclean, E., Mometto, N.: The Imandra Automated Reasoning System. In: Proc. of IJCAR (2020), to appear
19. Reger, G., Voronkov, A.: Induction in Saturation-Based Proof Search. In: Proc. of CADE. pp. 477–494 (2019)
20. Reger, G., Voronkov, A.: Induction in Saturation-Based Proof Search. EasyChair Smart Slide (2020), `https://easychair.org/smart-slide/slide/hXmP`
21. Reynolds, A., Blanchette, J.C.: A Decision Procedure for (Co)datatypes in SMT Solvers. In: Proc. of IJCAI. pp. 4205–4209 (2016)
22. Reynolds, A., Kuncak, V.: Induction for SMT Solvers. In: Proc. of VMCAI. pp. 80–98 (2015)
23. Rybina, T., Voronkov, A.: A Decision Procedure for Term Algebras with Queues. ACM Trans. Comput. Log. **2**(2), 155–181 (2001)
24. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An Automated Prover for Properties of Recursive Data Structures. In: Proc. of TACAS. pp. 407–421 (2012)
25. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A Cross-Community Infrastructure for Logic Solving. In: Proc. of IJCAR. pp. 367–373 (2014)
26. Voronkov, A.: AVATAR: The Architecture for First-Order Theorem Provers. In: Proc. of CAV. pp. 696–710. Springer-Verlag (2014)