Getting Saturated with Induction

Márton Hajdu[1], Petra Hozzová[1](✉), Laura Kovács[1], Giles Reger[2], and Andrei Voronkov[1,2,3]
{marton.hajdu, petra.hozzova}@tuwien.ac.at

[1] TU Wien
[2] University of Manchester
[3] EasyChair

Abstract. Induction in saturation-based first-order theorem proving is a new exciting direction in the automation of inductive reasoning. In this paper we survey our work on integrating induction directly into the saturation-based proof search framework of first-order theorem proving. We describe our induction inference rules proving properties with inductively defined datatypes and integers. We also present additional reasoning heuristics for strengthening inductive reasoning, as well as for using induction hypotheses and recursive function definitions for guiding induction. We present exhaustive experimental results demonstrating the practical impact of our approach as implemented within Vampire.

Keywords: Induction · Formal Verification · Theorem Proving

1 Introduction

One commonly used theory in the development of imperative/functional programs is the theory of inductively defined data types, such as natural numbers (e.g. see Figure 1(a)). Automating reasoning in formal verification therefore also needs to automate induction. Previous works on automating induction mainly focus on inductive theorem proving [3,4,5,22,17]: deciding when induction should be applied and what induction axiom should be used. Recent advances related to automating inductive reasoning, such as first-order reasoning with inductively defined data types [14], inductive strengthening [19] and structural induction in superposition [13,6,18,9,8], open up new possibilities for automating induction. In this paper we survey our recent results towards automating inductive reasoning for first-order properties with inductively defined data types and beyond.
Relation to the state-of-the-art. Our work automates induction by integrating it directly in the saturation-based approach of first-order provers [15,21,25]. These provers implement saturation-based proof search using the superposition calculus [16]. Moreover, they rely on powerful indexing algorithms, notions of redundancy, selection functions and term orderings for making theorem proving efficient. First-order theorem provers complement SMT solvers in reasoning with theories and quantifiers, as evidenced in the annual system competitions of SMT solvers [2,24] and first-order provers [23].

Our approach towards automating induction is conceptually different from previous attempts to use induction with superposition [13,6,8], as we are not restricted to specific clause splitting algorithms and heuristics [6], nor are we limited to induction over inductively defined data types using a subterm ordering [8]. As a result, we stay within the standard saturation framework and do not have to introduce constraint clauses, additional predicates, nor change the notion of redundancy as in [8]. In addition, our approach can be used to automate induction over arbitrary, and not just inductively defined, data types, such as integers (Section 8). Our work is also fundamentally different from rewrite-based approaches automating induction [3,4,17,5,19,22], as we do not rely on external algorithms/heuristics to generate subgoals/lemmas of an inductive property. Instead, applications of induction become inference rules of the saturation process, adding instances of appropriate induction schemata. We extend superposition reasoning with new inference rules capturing inductive steps (Sections 5-7), and optimize the saturation theorem proving process with induction. In addition, we instantiate induction axioms with logically stronger versions of the property being proved and use induction hypotheses as specialized rewrite rules (Section 8).

This combination of saturation with induction is very powerful. Our experimental results show that many problems previously unsolved by any system can be solved by our work, some resulting in very complex proofs of program properties and proofs of complex mathematical properties (Section 9).

Contributions. This paper serves as *a survey* of our recent progress in automating induction using a first-order theorem prover [18,9,12,11].

- We give a small tutorial of induction in saturation, helping non-experts in theorem proving to understand and further use our methodology. To this end, we describe saturation theorem proving and the main concepts of saturation with induction (Sections 4-5).
- We overview technical considerations for turning saturation with induction into an efficient approach (Section 5) and discuss variants of induction inference rules (Section 6).
- We present extensions of induction inference rules with multiple premises (Section 7), generalizations and integer reasoning (Section 8).
- We report on exhaustive experiments comparing and analysing our approach to state-of-the-art methods (Section 9).

2 Motivating Example

We motivate the challenges of automating induction for formal verification using the functional program of Figure 1(a). This program defines the inductively defined data type `nat` of natural numbers. In first-order logic, this data type corresponds to a term algebra with constructors `0` (zero) and `s` (successor); inductively defined data types, such as `nat`, are special cases of term algebras. The functional program in Figure 1(a) implements `add`, `even` and `half` operations over naturals, by using recursive equations (function definitions) preceded by

$\underline{\textbf{assume}}\ \ \texttt{even}(x)$

$\textbf{datatype}\ \ \texttt{nat} = \texttt{0} \mid \texttt{s}(x)$

$\textbf{fun}\ \texttt{add}(\texttt{0}, y) = y$
$\quad \mid \texttt{add}(\texttt{s}(z), y) = \texttt{s}(\texttt{add}(z, y));$

$\textbf{fun}\ \texttt{even}(\texttt{0}) = \top$
$\quad \mid \texttt{even}(\texttt{s}(z)) = \neg\texttt{even}(z);$

$\textbf{fun}\ \texttt{half}(\texttt{0}) = \texttt{0}$
$\quad \mid \texttt{half}(\texttt{s}(\texttt{0})) = \texttt{0}$
$\quad \mid \texttt{half}(\texttt{s}(\texttt{s}(z))) = \texttt{s}(\texttt{half}(z));$

$\underline{\textbf{assert}}\ \ x = \texttt{add}(\texttt{half}(x), \texttt{half}(x))$

(a)

Axiomatization of $\texttt{add}$, $\texttt{even}$ and $\texttt{half}$:

$\forall y \in \texttt{nat}.(\texttt{add}(\texttt{0}, y) = y)$

$\forall z, y \in \texttt{nat}.(\texttt{add}(\texttt{s}(z), y) = \texttt{s}(\texttt{add}(z, y)))$

$\texttt{even}(\texttt{0})$

$\forall z \in \texttt{nat}.(\texttt{even}(\texttt{s}(z)) \leftrightarrow \neg\texttt{even}(z))$

$\texttt{half}(\texttt{0}) = \texttt{0}$

$\texttt{half}(\texttt{s}(\texttt{0})) = \texttt{0}$

$\forall z \in \texttt{nat}.(\texttt{half}(\texttt{s}(\texttt{s}(z))) = \texttt{s}(\texttt{half}(z)))$

Verification task (conjecture):

$\forall x \in \texttt{nat}.(\texttt{even}(x) \rightarrow x = \texttt{add}(\texttt{half}(x), \texttt{half}(x)))$

(b)

Fig. 1. Motivating example over inductively defined data types.

the **fun** construct. These recursive equations correspond to universally quantified equalities in first-order logic, as listed in the axioms of Figure 1(b).

The expected behaviour of Figure 1(a) is specified using program assertions in first-order logic: the pre-condition using the **assume** construct and the post-condition using **assert**. Figure 1(a) satisfies its requirements. Formally proving correctness of Figure 1(a) essentially requires proving the conjecture of Figure 1(b), establishing that $\texttt{half}(x)$ of an $\texttt{even}$ natural number x added to $\texttt{half}(x)$ equals the original number x. That is,

$$\forall x \in \texttt{nat}.\big(\texttt{even}(x) \rightarrow x = \texttt{add}(\texttt{half}(x), \texttt{half}(x))\big). \tag{1}$$

Proving (1), and thus establishing correctness of Figure 1(a), is however challenging as it requires induction over the naturals. As such, finding and using an appropriate induction schemata is needed. The following sound *structural induction schema* for a formula F could, for example, be used, where F contains (multiple occurrences of) a natural-valued variable x:

$$\Big(F[\texttt{0}] \wedge \forall z \in \texttt{nat}.(F[z] \rightarrow F[\texttt{s}(z)])\Big) \rightarrow \forall x \in \texttt{nat}.F[x] \tag{2}$$

We instantiate schema (2) by considering $\forall x \in \texttt{nat}.F(x)$ to be formula (1), yielding the induction formula:

$$\begin{aligned}&\texttt{(IB)}\ \big(\texttt{even}(\texttt{0}) \rightarrow \texttt{0} = \texttt{add}(\texttt{half}(\texttt{0}), \texttt{half}(\texttt{0}))\big)\wedge\\ &\texttt{(IS)}\ \forall z \in \texttt{nat}.\left(\begin{array}{c}\big(\texttt{even}(z) \rightarrow z = \texttt{add}(\texttt{half}(z), \texttt{half}(z))\big) \rightarrow\\ \big(\texttt{even}(\texttt{s}(z)) \rightarrow \texttt{s}(z) = \texttt{add}(\texttt{half}(\texttt{s}(z)), \texttt{half}(\texttt{s}(z)))\big)\end{array}\right)\\ &\qquad \rightarrow \forall x \in \texttt{nat}.\texttt{even}(x) \rightarrow x = \texttt{add}(\texttt{half}(x), \texttt{half}(x)),\end{aligned} \tag{3}$$

where the subformulas denoted by (IB) and (IS) correspond to the *induction base case* and the *induction step case* of (3). Since schema (2) is sound, its instance (3) is valid. As such, the task of proving (1) is reduced to proving the base case and step case of (3).

Using the definitions of `half` and `add` from Figure 1(b), the base case `(IB)` simplifies to the tautology $\top \rightarrow \mathtt{0} = \mathtt{0}$. On the other hand, proving `(IS)` requires additional inductive reasoning. Yet, the induction scheme (2) cannot be used as $\mathtt{even}(z)$ and $\mathtt{even}(\mathtt{s}(z))$ yield two different base cases. We overcome this limitation by using an additional induction schema with two base cases, as follows:

$$\big(F[\mathtt{0}] \land F[\mathtt{s}(\mathtt{0})] \land \forall z.(F[z] \rightarrow F[\mathtt{s}(\mathtt{s}(z))])\big) \rightarrow \forall x.F[x] \tag{4}$$

As before, by instantiating (4) with (1) and simplifying based on the axioms of Figure 1(b), we are left with proving the step case:

$$\begin{array}{ll} \mathtt{(IH)} & \forall z \in \mathtt{nat.}\Big(\big(\mathtt{even}(z) \rightarrow z = \mathtt{add}(\mathtt{half}(z), \mathtt{half}(z))\big) \rightarrow \\ \mathtt{(IC)} & \qquad \big(\mathtt{even}(\mathtt{s}(\mathtt{s}(z))) \rightarrow \mathtt{s}(\mathtt{s}(z)) = \mathtt{add}(\mathtt{half}(\mathtt{s}(\mathtt{s}(z))), \mathtt{half}(\mathtt{s}(\mathtt{s}(z))))\big)\Big) \end{array} \tag{5}$$

The antecedent `(IH)` and conclusion `(IC)` of (5) are called the *induction (step) hypothesis* and *induction step conclusion* of the step case, respectively. After rewriting $\mathtt{even}(\mathtt{s}(\mathtt{s}(z)))$ to $\mathtt{even}(z)$ in `(IC)`, both `(IH)` and `(IC)` have the same assumption $\mathtt{even}(z)$, which can be discarded. By rewriting the remaining conclusions in `(IH)` and `(IC)` using the definitions of `half` and `add`, as well as the the injectivity of the term algebra constructor `s`, we obtain:

$$\begin{array}{ll} \mathtt{(IH)} & \forall z \in \mathtt{nat.}\big(z = \mathtt{add}(\mathtt{half}(z), \mathtt{half}(z)) \rightarrow \\ \mathtt{(IC)} & \qquad \mathtt{s}(z) = \mathtt{add}(\mathtt{half}(z), \mathtt{s}(\mathtt{half}(z)))\big) \end{array} \tag{6}$$

Since the more complex right-hand side of `(IH)` is not equal to any subterm of `(IC)` in (6), we have to use `(IH)` in the left-to-right direction – in order to preserve validity, our only option is to rewrite z on the left-hand side of `(IC)`:

$$\forall z \in \mathtt{nat.}\big(\mathtt{s}(\mathtt{add}(\mathtt{half}(z), \mathtt{half}(z))) = \mathtt{add}(\mathtt{half}(z), \mathtt{s}(\mathtt{half}(z)))\big) \tag{7}$$

Equation (7) is a special case of the formula $\forall x, y \in \mathtt{nat.s}(\mathtt{add}(x, y)) = \mathtt{add}(x, \mathtt{s}(y))$ which can be easily verified using the induction schema (2). This establishes the correctness of Figure 1(a).

The verification task of Figure 1(a) highlights the main difficulties in automating inductive reasoning: (i) incorporating induction into saturation (Section 5); (ii) finding suitable induction schemata (Section 6); and (iii) using extensions of induction inference rules to further push the boundaries of automating induction (Sections 7–8). We next present our solutions to these challenges, based on our results from [18,9,12,11].

3 Preliminaries

We assume familiarity with *standard multi-sorted first-order logic with equality.* Functions are denoted with f, g, h, predicates with p, q, r, variables with x, y, z, w, and Skolem constants with σ, all possibly with indices. A term is *ground* if it contains no variables. We use the words *sort* and *type* interchangeably. We distinguish special sorts called *term algebra sorts*, function symbols for term algebra sorts called *constructors* and *destructors*. For a term algebra sort τ, we denote its constructors with Σ_τ. For each $c \in \Sigma_\tau$, we denote its arity with n_c and the corresponding destructor returning the value of the ith argument of c by d^i_c. Moreover, we denote with P_c the set of argument positions of c of the sort τ. We say that c is a *recursive constructor* if P_c is non-empty, otherwise it is called a *base constructor*. We call the ground terms built from the constructor symbols of a sort its *term algebra*. We axiomatise term algebras using their *injectivity*, *distinctness*, *exhaustiveness* and *acyclicity* axioms [14]. We refer to term algebras also as algebraic data types or inductively defined data types. Additionally, we assume a distinguished *integer sort*, denoted by $\mathbb{Z}$. When we use standard integer predicates $<, \leq, >, \geq$, functions $+, -, \dots$ and constants $0, 1, \dots$, we assume that they denote the corresponding interpreted integer predicates and functions with their standard interpretations. All other symbols are uninterpreted.

We use the standard logical connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$, and quantifiers $\forall$ and $\exists$. We write quantifiers like $\forall x \in \tau$ to denote that x has the sort τ where it is not clear from the context. A *literal* is an atom or its negation. For a literal L, we write $\overline{L}$ to denote its complementary literal. A disjunction of literals is a *clause*. We denote clauses by C, D and reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. We denote the *clausal normal form* of a formula F by $\texttt{cnf}(F)$. We call every term, literal, clause or formula an *expression*.

We write $E[s]$ to denote that the expression E contains k distinguished occurrence(s) of the term s, with $k \geq 0$. For simplicity, $E[t]$ means that these occurrences of s are replaced by the term t. A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two terms s and t if $s\theta = t\theta$, and is a *most general unifier* (*mgu*) if for every unifier η of s and t, there exists substitution μ s.t. $\eta = \theta\mu$. We denote the mgu of s and t with $\texttt{mgu}(s, t)$.

4 Saturation-Based Theorem Proving

We briefly introduce saturation-based proof search, which is the leading technology for automated first-order theorem proving. For details, we refer to [15].

First-order theorem provers work with clauses, rather than with arbitrary formulas. Given a set S of input clauses, first-order provers *saturate* S by computing all logical consequences of S with respect to a sound inference system $\mathcal{I}$. The saturated set of S is called the *closure* of S and the process of computing the closure of S is called *saturation*. If the closure of S contains the empty clause $\square$, the original set S of clauses is unsatisfiable. A simplified saturation algorithm for a sound inference system $\mathcal{I}$ is given in Algorithm 1, with a clausified goal B ($\neg B$ is also clausified) and clausified assumptions A as input.

Note that a saturation algorithm proves validity of B by establishing unsatisfiabiliy of $\neg B$ using the assumptions A; we refer to this proving process as a *refutation* of $\neg B$ from A. Completeness and efficiency of saturation-based reasoning rely heavily on properties of selection and addition of clauses from/to

Algorithm 1 The Saturation Loop.

```
1 initial set of clauses $S := A \cup \{\neg B\}$
2 repeat
3   Select clause $G \in S$
4   Derive consequences $C_1, \ldots, C_n$ of $G$ and formulas from $S$ using rules of $\mathcal{I}$
5   $S := S \cup \{C_1, \ldots, C_n\}$
6   if $\Box \in S$ then return $A \to B$ is UNSAT
8 return $A \to B$ is SAT
```

Superposition:

$$\frac{l = r \lor C \quad L[l'] \lor D}{(L[r] \lor C \lor D)\theta} \qquad \frac{l = r \lor C \quad s[l'] \neq t \lor D}{(s[r] \neq t \lor C \lor D)\theta} \qquad \frac{l = r \lor C \quad s[l'] = t \lor D}{(s[r] = t \lor C \lor D)\theta}$$

where $\theta := \texttt{mgu}(l, l')$, $r\theta \not\succeq l\theta$, (first rule only) $L[l']$ is not an equality literal, and (second and third rules only) $t\theta \not\succeq s[l']\theta$.

Binary resolution:

$$\frac{L \lor C \quad \neg L' \lor D}{(C \lor D)\theta}$$

where $\theta := \texttt{mgu}(L, L')$.

Equality resolution:

$$\frac{s \neq t \lor C}{C\theta}$$

where $\theta := \texttt{mgu}(s, t)$.

Equality factoring:

$$\frac{s = t \lor s' = t' \lor C}{(s = t \lor t \neq t' \lor C)\theta}$$

where $\theta := \texttt{mgu}(s, s')$, $t\theta \not\succeq s\theta$, and $t'\theta \not\succeq t\theta$.

Fig. 2. The superposition calculus $\mathbb{S}\text{up}$ for first-order logic with equality.

S, using the inference system $\mathcal{I}$ (lines 3–5). To organize saturation, first-order provers use simplification *orderings* on terms, which are extended to orderings over literals and clauses; for simplicity, we write $\succ$ for both the term ordering and its clause/multiset ordering extensions. Given an ordering $\succ$, a clause C is *redundant* with respect to a set S of clauses if there exists a subset S' of S such that S' is smaller than $\{C\}$, that is $\{C\} \succ S'$ and $S' \to C$.

The *superposition calculus*, denoted as $\mathbb{S}\text{up}$, is the most common inference system employed by saturation-based first-order theorem provers for first-order logic with equality [16]. A summary of superposition inference rules is given in Figure 2. The superposition calculus $\mathbb{S}\text{up}$ is *sound* and *refutationally complete*: for any unsatisfiable formula $\neg B$, the empty clause can be derived as a logical consequence of $\neg B$.

5 Saturation with Induction

We now describe our approach towards automating inductive reasoning within saturation-based proof search. We illustrate the key ingredients of our method using our motivating example from Figure 1(a), that is proving (1) in order to establish correctness of Figure 1(a). As mentioned in Section 4, proving (1) in a saturation-based approach means refuting the clausified negation of (1), that is, refuting the following two clauses:

$$\texttt{even}(\sigma_0) \tag{8}$$

$$\sigma_0 \neq \texttt{add}(\texttt{half}(\sigma_0), \texttt{half}(\sigma_0)) \tag{9}$$

We establish invalidity of inductive formulas, such as (8)-(9), by *integrating the application of induction as additional inference rules of the saturation process*. Our induction inference rules are used directly in Algorithm 1, as follows:

(i) we pick up an inductive property G in the search space S (line 3);
(ii) derive new induction axioms $C_1, \ldots, C_n$ (instances of *induction schemata*), aiming at refuting G, or sometimes a more general formula than G (line 4);
(iii) add the induction axioms $C_1, \ldots, C_n$ to the search space (line 5).

Our work therefore follows a different approach than the one used in inductive theorem provers, as we do not rely on external algorithms to generate subgoals/stronger formulas G' of an inductive property G nor do we replace G by subgoals/stronger formulas G'. Rather, new induction axioms C_i, and sometimes new induction axioms C_i' for more general formulas G', are derived from G and used in the search space S *in addition* to G.

Finding the right induction schema and developing efficient induction inference rules for deriving inductive axioms/formulas (steps (i)-(ii) above) are crucial for saturation with induction. In [18] we introduced the following induction inference rule, parametrized by a valid induction schema:

$$\frac{\overline{L}[t] \vee C}{\texttt{cnf}(F \rightarrow \forall x.L[x])}\ (\texttt{Ind}),$$

where t is a ground term, L is a ground literal, C is a clause, and $F \rightarrow \forall x.L[x]$ is a valid induction schema. For example, the induction schema (2) for F can be used in (Ind). We call $\overline{L}[t]$ the *induction literal* and t the *induction term*. We note that (Ind) can naturally be generalized to handle multiple induction terms, as in [11]. In this paper, we only use the rule with one induction term.

Based on Algorithm 1 (the saturation-based proof search algorithm), note that the application of (Ind) adds new clauses to the search space by clausifying induction formulas (cnf() in (Ind)). These new clauses then become potential candidates to be selected in the next steps of the algorithm. As such, the selection of these new clauses are likely to be delayed, and thus their use in proving an inductive goal becomes highly inefficient. We therefore propose the application of (Ind) followed by a binary resolution step to "guide" induction over selected induction literals and terms. In particular, upon the application of (Ind), we do not add $\texttt{cnf}(F \rightarrow \forall x.L[x])$ to the search space. Instead, we binary resolve the conclusion literal $L[x]$ against $\overline{L}[t]$, allowing us to only add the formula $\texttt{cnf}(\neg F) \vee C$ to the search space, whenever (Ind) is applied.

In order to "guide" and combine the application of (Ind) with a binary resolution rule, we exploit instances of (Ind) for special cases of induction schemata over term algebras (Section 6) and integers (Section 8). We also consider extension of (Ind) for more general and efficient inductive reasoning (Section 7–8).

6 Induction with Term Algebras

We first consider the theory of term algebras and introduce instances of the induction rule (Ind), by exploiting properties of the induction literal $\overline{L}[t]$ and induction schemata over the induction term t. For now, the induction term t is a ground element from a term algebra.

Structural Induction. The first instance of (Ind) uses the following constructor-based structural induction schema, where $L[x]$ is a literal containing (possibly multiple occurrences of) x of a term algebra sort τ:

$$\Big(\bigwedge_{c\in\Sigma_\tau} \forall y_1,...,y_{n_c}.(\wedge_{i\in P_c} L[y_i] \to L[c(y_1,...,y_{n_c})])\Big) \to \forall x\in\tau.L[x] \tag{10}$$

Note that the structural induction schema (2) over naturals is an instance of (10).

Example 1. By instantiating schema (10) with the sole literal of clause (9) and induction term σ_0, we obtain:

$$\left(\begin{array}{c} \mathtt{0} = \mathtt{add}(\mathtt{half}(\mathtt{0}),\mathtt{half}(\mathtt{0})) \wedge \\ \forall z\in\mathtt{nat}.\left(\begin{array}{c} z = \mathtt{add}(\mathtt{half}(z),\mathtt{half}(z)) \to \\ \mathtt{s}(z) = \mathtt{add}(\mathtt{half}(\mathtt{s}(z)),\mathtt{half}(\mathtt{s}(z)))\end{array}\right)\end{array}\right) \to \begin{array}{c}\forall x\in\mathtt{nat}.\big(x = \\ \mathtt{add}(\mathtt{half}(x),\mathtt{half}(x))\big)\end{array} \tag{11}$$

The clausified form of (11) consists of the following two clauses:

$$\mathtt{0} \neq \mathtt{add}(\mathtt{half}(\mathtt{0}),\mathtt{half}(\mathtt{0})) \vee \sigma_1 = \mathtt{add}(\mathtt{half}(\sigma_1),\mathtt{half}(\sigma_1)) \vee x = \mathtt{add}(\mathtt{half}(x),\mathtt{half}(x))$$
$$\mathtt{0} \neq \mathtt{add}(\mathtt{half}(\mathtt{0}),\mathtt{half}(\mathtt{0})) \vee \mathtt{s}(\sigma_1) \neq \mathtt{add}(\mathtt{half}(\mathtt{s}(\sigma_1)),\mathtt{half}(\mathtt{s}(\sigma_1))) \\ \vee x = \mathtt{add}(\mathtt{half}(x),\mathtt{half}(x))$$

After applying (Ind) instantiated with (11) on (9), the above clauses are resolved with the literal in clause (9), adding to the search space the resulting clauses:

$$\mathtt{0} \neq \mathtt{add}(\mathtt{half}(\mathtt{0}),\mathtt{half}(\mathtt{0})) \vee \sigma_1 = \mathtt{add}(\mathtt{half}(\sigma_1),\mathtt{half}(\sigma_1))$$
$$\mathtt{0} \neq \mathtt{add}(\mathtt{half}(\mathtt{0}),\mathtt{half}(\mathtt{0})) \vee \mathtt{s}(\sigma_1) \neq \mathtt{add}(\mathtt{half}(\mathtt{s}(\sigma_1)),\mathtt{half}(\mathtt{s}(\sigma_1)))$$ □

Well-Founded Induction. Two other instances of (Ind) exploit well-founded induction schemata, by using a binary well-founded relation R on a term algebra τ. For such an R, if there does not exists a smallest value $v\in\tau$ w.r.t. R such that $L[v]$ does not hold, then $L[x]$ holds for any $x\in\tau$. This principle is formalized by the following schema:

$$\Big(\neg\exists y\in\tau.\big(\neg L[y] \wedge \forall z\in\tau.(R(y,z)\to L[z])\big)\Big) \to \forall x\in\tau.L[x] \tag{12}$$

However, to instantiate (12), we need to find an R suitable for the considered τ.

Similarly to [20], we first consider the direct subterm relation expressed using term algebra constructors and destructors of the term algebra sort τ. We obtain the following instance of (12) to be applied in (Ind):

$$\Big(\neg\exists y.\big(\neg L[y] \wedge \bigwedge_{c\in\Sigma_\tau}(y = c(d_c^1(y),\ldots,d_c^{n_c}(y)) \to \bigwedge_{i\in P_c} L[d_c^i(y)])\big)\Big) \to \forall x.L[x] \tag{13}$$

In the case of natural numbers, where `p` is the destructor for `s`, we have the following instance of (13) to be used in (`Ind`):

$$\Big(\neg\exists y \in \mathtt{nat}.\big(\neg L[y] \wedge (y = \mathtt{s}(\mathtt{p}(y)) \rightarrow L[\mathtt{p}(y)])\big)\Big) \rightarrow \forall x \in \mathtt{nat}.L[x] \qquad (14)$$

Another instance of (12) to be used in `(Ind)` employs a fresh predicate $\mathtt{less}_y$, as given next. The axiomatisation of such a predicate enables efficient reasoning over subterm properties withing saturation, as advocated in [14].

$$\begin{aligned}\Big(\neg\exists y.\big(\neg F[y] \wedge \forall z.(\mathtt{less}_y(z) \rightarrow F[z]) \wedge (y = \mathtt{s}(\mathtt{p}(y)) \rightarrow \mathtt{less}_y(\mathtt{p}(y)))\\ \wedge \forall w.(\mathtt{less}_y(\mathtt{s}(\mathtt{p}(w))) \rightarrow \mathtt{less}_y(\mathtt{p}(w)))\big)\Big) \rightarrow \forall x.F[x]\end{aligned} \qquad (15)$$

Induction with Recursive Function Definitions. In formalizing the induction schemata instances given e.g. in (2) and (14), we considered the term algebra `nat` as an instance of τ. To come up with the "right" term algebra instance of τ, we can also use terminating recursive function definitions from the input problem to be proven, such as `add`, `even` and `half` from Figure 1(a). The termination of such recursive functions naturally depends on a well-founded relation R.

Example 2. We can obtain schema (4) from `half` in Figure (1)(b) if we consider the well-founded relation based on its first argument. In particular, the third branch of `half` relates its first argument $\mathtt{s}(\mathtt{s}(z))$ to z in its recursive call for all $z \in \mathtt{nat}$. This relation gives the step case of schema (4), and the base cases can be obtained by considering the terms in the first argument positions for the other two branches of `half`.

Thus, based on the term $\mathtt{half}(\sigma_0)$ in clause (9), we can instantiate (4) inducting on term σ_0. However, this induction axiom does not yet lead to a refutation of (1), because for each clausified induction axiom, new Skolem constants are introduced. Thus, the literals in clauses resulting from applying `(Ind)` on (8) or (9), respectively, do not contain σ_0, and hence we cannot use (9) nor (8), respectively, to refute them. In the next section we therefore generalize `(Ind)` towards the use of induction schemata with multiple clauses. □

7 Multi-Clause Induction

Inducting on a single literal is sometimes not sufficient to get a refutation, as illustrated in Example 2 for Figure 1(a). In general however, induction can be applied on literals from multiple clauses, similarly to formula (3) in Section 2. We generalize the inference rule `(Ind)` towards multi-clause induction `(IndMC)`:

$$\frac{L_1[t] \vee C_1 \quad ... \quad L_n[t] \vee C_n \quad \overline{L}[t] \vee C}{\mathtt{cnf}(F \rightarrow \forall x.(\bigwedge_{1\leq i\leq n} L_i[x] \rightarrow L[x]))}\ (\mathtt{IndMC})$$

where $F \rightarrow \forall x.(\bigwedge_{1\leq i\leq n} L_i[x] \rightarrow L[x])$ is a valid induction formula, $\overline{L}$ and L_i are ground literals and C and C_i are clauses. Similarly to (`Ind`), our new rule `(IndMC)` is used within saturation-based proof as an additional inference rule, followed by an application of binary resolution for guiding inductive reasoning.

Example 3. We use schema (4) with formula (1) with induction term σ_0 to instantiate `(IndMC)` for premises (8) and (9). The induction formula is:

$$\begin{pmatrix} \big(\mathtt{even}(\mathtt{0}) \to \mathtt{0} = \mathtt{add}(\mathtt{half}(\mathtt{0}),\mathtt{half}(\mathtt{0}))\big)\wedge \\ \big(\mathtt{even}(\mathtt{s}(\mathtt{0})) \to \mathtt{s}(\mathtt{0}) = \mathtt{add}(\mathtt{half}(\mathtt{s}(\mathtt{0})),\mathtt{half}(\mathtt{s}(\mathtt{0})))\big)\wedge \\ \forall z{\in}\mathtt{nat}.\begin{pmatrix} \big(\mathtt{even}(z) \to z = \mathtt{add}(\mathtt{half}(z),\mathtt{half}(z))\big) \to \\ \big(\mathtt{even}(\mathtt{s}(\mathtt{s}(z))) \to \mathtt{s}(\mathtt{s}(z)) = \mathtt{add}(\mathtt{half}(\mathtt{s}(\mathtt{s}(z))),\mathtt{half}(\mathtt{s}(\mathtt{s}(z))))\big)\end{pmatrix}\end{pmatrix} \\ \to \forall x \in \mathtt{nat}.\big(\mathtt{even}(x) \to x = \mathtt{add}(\mathtt{half}(x),\mathtt{half}(x))\big) \tag{16}$$

Clausification of formula (16) results in twelve clauses, each containing the literals $\neg\mathtt{even}(x)$ and $x = \mathtt{add}(\mathtt{half}(x),\mathtt{half}(x))$, which we can binary resolve with clauses (8) and (9). After simplifications are applied to the clauses from formula (16), we are left with the following two clauses:

$$\sigma_2 = \mathtt{add}(\mathtt{half}(\sigma_2),\mathtt{half}(\sigma_2)) \tag{17}$$

$$\mathtt{s}(\sigma_2) \neq \mathtt{add}(\mathtt{half}(\sigma_2),\mathtt{s}(\mathtt{half}(\sigma_2))) \tag{18}$$

We now need to rewrite (18) with the induction hypothesis clause (17) in the left-to-right orientation. However, $\sigma_2 \prec \mathtt{add}(\mathtt{half}(\sigma_2),\mathtt{half}(\sigma_2))$, which holds for any simplification ordering $\prec$, contradicts the superposition ordering conditions. Moreover, even if we rewrote against the ordering, we would be left with

$$\mathtt{s}(\mathtt{add}(\mathtt{half}(\sigma_2),\mathtt{half}(\sigma_2))) \neq \mathtt{add}(\mathtt{half}(\sigma_2),\mathtt{s}(\mathtt{half}(\sigma_2))), \tag{19}$$

which is hard to refute using induction due to the induction term σ_2 occurring in the second argument of `add`, which does not change in the recursive definition of `add` (see Figure 1(b)). We overcome this limitation by extensions of inductive reasoning in Section 8. □

8 Extensions of Inductions in Saturation

Induction with Generalizations. It is common in mathematics that for proving a formula A, we prove instead a formula B such that $B \to A$. In other words, we prove a *generalization* B of A. Inductive theorem provers implement various heuristics to *guess formulas/lemmas* B and use B instead of A during proof search, see e.g. [4,5,3,17]. However, a saturation-based theorem prover would not/can not do this, since goals/conjectures are not replaced by sub-goals in saturation-based proof search. We thus propose a different approach for implementing the common generalization recipe of mathematical theorem proving. Namely, we introduce the inference rule `(IndGen)` of *induction with generalization*, allowing us to (i) add instances of induction schemata not only for A but also for versions of B and then (ii) perform saturation over these induction schemata instances, using superposition reasoning. Our `(IndGen)` rule inducts only on *some* occurrences of the induction term t, as follows:

$$\frac{\overline{L}[t] \vee C}{\mathtt{cnf}(F \to \forall x.L'[x])}\ (\mathtt{IndGen}),$$

where t is a ground term, L is a ground literal, C is a clause, $F \rightarrow \forall x.L'[x]$ is a valid induction schema and $L'[x]$ is obtained from $L[t]$ by replacing some occurrences of t with x.

Example 4. We illustrate induction with generalization on the unit clause (19). One generalization that would help refute (19) by eliminating $\mathtt{half}(\sigma_2)$ is:

$$\forall x, y \in \mathtt{nat}.\mathtt{s}(\mathtt{add}(x, y)) = \mathtt{add}(x, \mathtt{s}(y)) \tag{20}$$

Instantiating schema (2) with (20) and variable x would lead to a refutation when used with rule (IndGen) on (19). However, since we do not use y from the generalization in the induction, there is no need to replace the occurrences of $\mathtt{half}(\sigma_2)$ corresponding to it in the generalized literal. Our final generalized induction formula, also leading to the refutation of (19), is:

$$\begin{pmatrix} \mathtt{s}(\mathtt{add}(0, \mathtt{half}(\sigma_2))) = \mathtt{add}(0, \mathtt{s}(\mathtt{half}(\sigma_2))) \wedge \\ \forall z \in \mathtt{nat}. \begin{pmatrix} \mathtt{s}(\mathtt{add}(z, \mathtt{half}(\sigma_2))) = \mathtt{add}(z, \mathtt{s}(\mathtt{half}(\sigma_2))) \rightarrow \\ \mathtt{s}(\mathtt{add}(\mathtt{s}(z), \mathtt{half}(\sigma_2))) = \mathtt{add}(\mathtt{s}(z), \mathtt{s}(\mathtt{half}(\sigma_2))) \end{pmatrix} \end{pmatrix} \tag{21}$$
$$\rightarrow \forall x \in \mathtt{nat}.\mathtt{s}(\mathtt{add}(x, \mathtt{half}(\sigma_2))) = \mathtt{add}(x, \mathtt{s}(\mathtt{half}(\sigma_2))) \qquad \square$$

Rewriting with Induction Hypotheses. For turning saturation-based proof search into an efficient process, one key ingredient is to ensure that bigger terms/literals are rewritten by small ones (big/small w.r.t. the simplification ordering $\succ$), and not vice versa. However, this often prohibits using induction hypotheses to rewrite their corresponding conclusions which would be the necessary step to proceed with the proof. To overcome this obstacle, we introduce the following inference rule which uses an induction hypothesis literal to rewrite its conclusion:

$$\frac{l = r \vee D \quad s[l] \neq t \vee C}{\mathtt{cnf}(F \rightarrow \forall x.(s[r] = t)[x])} \; (\mathtt{IndHRW})$$

where $s[l] \neq t$ is an induction conclusion literal with corresponding induction hypothesis literal $l = r$, $l \not\succeq r$, and $F \rightarrow \forall x.(s[r] = t)[x]$ is a valid induction formula. Moreover, we resolve the clauses with the intermediate clause $s[r] \neq t \vee C \vee D$, obtained from the rewriting of the premises of ($\mathtt{IndHRW}$).

Example 5. Using unit clause (17) in a left-to-right orientation and rewriting the sides of unit clause (18) one after the other, we get intermediate clauses, which are then used for generating induction formulas. One such intermediate clause is (19), from which the induction formula (21) is generated. After clausifying (21), a subsequent binary resolution is performed with intermediate clause (19). By more simplifications using the definition of add and the injectivity of s, we finally obtain a refutation of (1), concluding thus the correctness of Figure 1(a). $\square$

Integer Induction. The last extension of our induction framework we introduce is *integer induction*, motivated by the need of inductive reasoning in program analysis and verification problems using integers. As the standard order $<$ (or

$>$) over integers $\mathbb{Z}$ is not well-founded, we work with *subsets of* $\mathbb{Z}$ *with a lower (and/or an upper) bound*. We therefore define the *downward, respectively upward, induction schema with symbolic bound* b as any formula of the form:

$$F[b] \wedge \forall y \in \mathbb{Z}.(y \leq b \wedge F[y] \rightarrow F[y-1]) \rightarrow \forall x \in \mathbb{Z}.(x \leq b \rightarrow F[x]); \quad \textit{(downward)}$$
$$F[b] \wedge \forall y \in \mathbb{Z}.(y \geq b \wedge F[y] \rightarrow F[y+1]) \rightarrow \forall x \in \mathbb{Z}.(x \geq b \rightarrow F[x]), \quad \textit{(upward)}$$

respectively, where $F[x]$ is a formula with one or more occurrences of an integer variable x and b is an integer term not containing x nor y. Further, we also define *interval downward, respectively upward, induction schema with symbolic bounds* b_1, b_2 as any formula of the form:

$$F[b_2] \wedge \forall y \in \mathbb{Z}.(b_1 < y \leq b_2 \wedge F[y] \rightarrow F[y-1]) \rightarrow \forall x \in \mathbb{Z}.(b_1 \leq x \leq b_2 \rightarrow F[x]); \quad \textit{(down.)}$$
$$F[b_1] \wedge \forall y \in \mathbb{Z}.(b_1 \leq y < b_2 \wedge F[y] \rightarrow F[y+1]) \rightarrow \forall x \in \mathbb{Z}.(b_1 \leq x \leq b_2 \rightarrow F[x]), \quad \textit{(up.)}$$

respectively, where $F[x]$ is a formula with one or more occurrences of an integer variable x and b_1, b_2 are integer terms not containing x nor y.[4]

To automate inductive reasoning over integers, we need to automatically generate suitable instances of our integer induction schemata. To this end we introduce induction rules with the integer induction schemata in the conclusion, giving us the recipe for instantiating the schemata. Since our schemata are sound, all resulting induction rules are sound as well. When t, b are ground terms and $L[t]$ is a ground literal, the following is an *integer upward induction rule*:

$$\frac{\overline{L}[t] \vee C \qquad t \geq b}{\texttt{cnf}\Big(\big(L[b] \wedge \forall y \in \mathbb{Z}.(y \geq b \wedge L[y] \rightarrow L[y+1])\big) \rightarrow \forall x \in \mathbb{Z}.(x \geq b \rightarrow L[x])\Big)} \ (\texttt{IntInd}_{\geq})$$

Our further integer induction rules using the other schemata are obtained similarly, as detailed in [12].

9 Implementation and Experiments

9.1 Implementation

Our approach for automating induction in saturation is implemented in the Vampire prover. All together, our implementation consists of around 7,800 lines of C++ code and is available online at `https://github.com/vprover/vampire/tree/int-induction`. In the following, Vampire* refers to the Vampire version supporting induction.

Our induction rules allow us to derive many new clauses potentially leading to refutation of inductive properties. These new clauses might however pollute the search space without advancing the proof. We therefore introduce options to control the use of induction rules by inducting only on negative literals, unit clauses or clauses derived from the goal. Further, for induction over algebraic types, we only allow induction on terms containing a constant other than a

[4] The above schemata can be seen as a special case of the multi-clause schemata used in the (IndMC) rule from Section 7, tailored specifically for integers.

Name & comma-separated values	Description
`--induction` `int, struct, both, none` (default: none)	Enable induction on integers only, or induction on algebraic types only, or both, or none
`--induction_on_complex_terms` `on, off` (default: off)	Apply induction also on complex terms
`--induction_multiclause` `on, off` (default: on)	Enable the `(IndMC)` form of induction rules
`--induction_gen` `on, off` (default: off)	Enable the `(IndGen)` form of induction rules

Table 1. Selected induction options or VAMPIRE. Default values are underlined.

base constructor. For integer induction, by default we disable rules with default bound, induction on interpreted constants, and induction on some comparison literals. Our most relevant induction options are summarized in Table 1.[5]

9.2 Experimental Setup

The main goal of our experiments was to evaluate how much induction improves VAMPIRE's performance. We therefore compared VAMPIRE* to VAMPIRE without induction. We also show the numbers of problems solved by the SMT solvers CVC4 [20], Z3 [7], where only CVC4 supports induction. In our experiments, we do not include other provers, such as ACL2 [3] or ZIPPERPOSITION [6], as these solvers do not support the SMT-LIB input format [1]; yet for further comparison we refer to [9,12,11].

We ran our experiments using (i) benchmarks over inductive data types (UFDT set of the SMT-LIB benchmark library and *dty* set of the inductive benchmarks [10]), (ii) benchmarks using integers (LIA, UFLIA, NIA and UFNIA of SMT-LIB and *int* of [10]), and (iii) benchmarks using both integers and data types (UFDTLIA of SMT-LIB). From these datasets, we excluded those problems that are marked satisfiable, as our work is meant for validity checking[6]

For our experiments, we used Z3 version 4.8.12 in the default configuration, and CVC4 version 1.8 with parameters `--conjecture-gen --quant-ind`. To extensively compare VAMPIRE and VAMPIRE*, we ran multiple instances of both for each experiment: we used a portfolio of 18 base configurations differing in the parameters not related to induction. Additionally, we varied the induction parameters of VAMPIRE* for each experiment: for (i) we used `--induction struct --structural_induction_kind one --induction_gen on -induction_on_complex_terms on`, for (ii) `--induction int --induction_multiclause off`, for (iii) `--induction both --structural_induction_kind one --induction_gen on -induction_on_complex_terms on`. In experiments (ii) and (iii), for each

[5] VAMPIRE also offers a so-called portfolio mode, in which it sequentially tries different option configurations for short amounts of time.

[6] we have excluded all together 1562 satisfiable problems from LIA, UFLIA, NIA and UFNIA; and 86 satisfiable problems from UFDT.

Problem	SMT-LIB						ind. set [10]		
set	UFDT	UFDTLIA	LIA	UFLIA	NIA	UFNIA	*dty*	*int*	sum
Total count	4483	327	404	10118	8	12181	3397	120	31038
Vampire	1848	82	241	6125	3	3704	17	0	12020
Vampire*	1792	186	241	6240	4	3679	464	76	12682
Cvc4	2072	200	357	6911	7	3022	164	30	12763
Z3	1807	76	242	6710	2	4938	17	0	13792

Table 2. Comparison of the number of solved problems. The configuration of Vampire and Vampire* depends on the benchmark set.

of the 18 base configurations we ran 7 instances of Vampire* with different integer induction parameters, chosen based on preliminary experimentation on a smaller set of benchmarks. Each prover configuration was given 10 seconds and 16 GB of memory per each problem. The experiments were ran on computers with 32 cores (AMD Epyc 7502, 2.5 GHz) and 1 TB RAM.

9.3 Experimental Results

Results overview. Our results are summarized in Table 2. For Vampire and Vampire* we show the number of problems solved by the most successful configuration. Note that for different benchmark sets the most successful configurations might be different. In the inductive problems, the maximum and average numbers of induction steps in a proof were 20 and 1.54, respectively, and the maximum number of nested induction steps was 9. Overall, Table 2 shows that Vampire* outperforms Vampire without induction. Moreover, Vampire* is competitive with leading SMT solvers.

Comparison of Vampire *and* Vampire**.* To evaluate the impact of inductive reasoning in Vampire, we look at two key metrics: the *overall number of solved problems*; and the *number of newly solved problems*, which we define as the number of problems solved using induction[7] by some Vampire*, but not solved by any Vampire. The latter metric is especially important, since in practice, one can run multiple solvers or configurations in parallel, and thereby solve the union of all problems solved by individual solvers.

Table 3 summarizes our result. Column "Combined" lists the number of problems solved by any instance of the configuration, and in the parentheses the number of problems newly solved by the configuration. The other columns (most solved, most new, default mode) give the numbers of solved problems, and in parentheses newly solved problems, for the corresponding Vampire/Vampire* instance. The "Default mode" columns shows results for the best induction configuration with all non-induction parameters set to default.

[7] New rules change proof search organization and Vampire* might solve a problem without using induction, while this problem was not solved by Vampire. We do not consider such problems to be newly solved.

Benchmarks	Configurations	Combined	Most solved	Most new	Default mode
UFDT	Vampire	2082	1848	-	1827
	Vampire*	2047	1792 (12)	1754 (17)	1761
dty	Vampire	17	17	-	17
	Vampire*	525	464 (453)	464 (453)	432
LIA, UFLIA, NIA, UFNIA	Vampire	11260	10073	-	9835
	Vampire*	11334 (81)	10051 (0)	9006 (41)	9773 (0)
int	Vampire	0	0	-	0
	Vampire*	118 (118)	76 (76)	76 (76)	49 (49)
UFDTLIA	Vampire	91	82	-	65
	Vampire*	197 (108)	186 (101)	186 (101)	136 (72)

Table 3. Comparison of Vampire and Vampire* configurations; numbers given (in parentheses) indicate new problems solved using induction but not without induction.

Induction helped most with the `dty`, `int` and UFDTLIA benchmark sets, as these sets contain a lot of problems focused on induction (induction was used in 91% of proofs for problems in `dty`, in all proofs in `int`, and in 71% of proofs in UFDTLIA), while the other sets contain a wide variety of problems (induction was only used in 2% of proofs in UFDT and 8.8% of proofs in LIA, UFLIA, NIA and UFNIA). Interestingly, the configuration which solved most problems in `int` solved the least in LIA, UFLIA, NIA, UFNIA combined, what illustrates the difficulty in choosing the right values for integer induction parameters for such a mixed benchmark set.

10 Conclusion

Motivated by application of program analysis and verification, we describe recent advances in automating inductive reasoning about first-order (program) properties using inductively defined data types and beyond. We integrate induction in the saturation-based proof engine of first-order theorem provers, without radical changes in the existing machinery of such provers. Our inductive inference rules and heuristics open up new research directions to be further studied in automating induction. Guiding and further extending the application of multi-clause induction with theory-specific induction schema variants is an interesting line of research. Combining induction schemas and rules and using lemma generation and rewriting procedures from inductive theorem provers are another ways to further improve saturation-based inductive reasoning.

Acknowledgements. We thank Johannes Schoisswohl for joint work related on experimenting with inductive theorem provers. This work was partially funded by the ERC CoG ARTIST 101002685, the EPSRC grant EP/P03408X/1, the FWF grant LogiCS W1255-N23, the Amazon ARA 2020 award FOREST and the TU Wien SecInt DK.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
2. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability modulo Theories Competition. In: Proceedings of the 17th International Conference on Computer Aided Verification. p. 20–23. CAV'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11513988_4
3. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. Academic Press (1988). https://doi.org/10.1016/C2013-0-10412-6
4. Bundy, A., Stevens, A., Harmelen, F.V., Ireland, A., Smaill, A.: Rippling: A heuristic for guiding inductive proofs. Artif. Intell. **62**, 185–253 (1993). https://doi.org/10.1016/0004-3702(93)90079-Q
5. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating Inductive Proofs Using Theory Exploration. In: Bonacina, M.P. (ed.) CADE. pp. 392–406. Springer (06 2013). https://doi.org/10.1007/978-3-642-38574-2_27
6. Cruanes, S.: Superposition with Structural Induction. In: Dixon, C., Finger, M. (eds.) FroCoS. pp. 172–188. Springer (2017)
7. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proc. of TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Echenheim, M., Peltier, N.: Combining Induction and Saturation-Based Theorem Proving. J. Automated Reasoning **64**, 253–294 (2020)
9. Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with Generalization in Superposition Reasoning. In: Benzmüller, C., Miller, B. (eds.) Proc. of CICM. LNCS, vol. 12236, pp. 123–137. Springer (2020). https://doi.org/10.1007/978-3-030-53518-6_8
10. Hajdu, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Inductive benchmarks for automated reasoning. In: Kamareddine, F., Sacerdoti Coen, C. (eds.) Proc. of CICM. pp. 124–129. Springer International Publishing, Cham (2021)
11. Hajdu, M., Hozzová, P., Kovacs, L., Voronkov, A.: Induction with recursive definitions in superposition. EasyChair Preprint no. 6513 (EasyChair, 2021)
12. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. In: Platzer, A., Sutcliffe, G. (eds.) CADE. pp. 361–377. Springer International Publishing, Cham (2021)
13. Kersani, A., Peltier, N.: Combining Superposition and Induction: A Practical Realization. In: Proc. of FroCoS. pp. 7–22 (2013)
14. Kovács, L., Robillard, S., Voronkov, A.: Coming to Terms with Quantified Reasoning. In: Castagna, G., Gordon, A.D. (eds.) POPL. pp. 260–270 (2017). https://doi.org/10.1145/3093333.3009887
15. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV. pp. 1–35. Springer (2013)
16. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chap. 7, pp. 371–443. North-Holland (2001)
17. Passmore, G.O., Cruanes, S., Ignatovich, D., Aitken, D., Bray, M., Kagan, E., Kanishev, K., Maclean, E., Mometto, N.: The Imandra Automated Reasoning System (System Description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR. pp. 464–471. Springer (2020). https://doi.org/10.1007/978-3-030-51054-1_30

18. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: Fontaine, P. (ed.) CADE. pp. 477–494. Springer (2019)
19. Reynolds, A., Kuncak, V.: Induction for SMT Solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI. pp. 80–98. Springer (2015). https://doi.org/10.1007/978-3-662-46081-8_5
20. Reynolds, A., Kuncak, V.: Induction for SMT Solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Proc. of VMCAI. LNCS, vol. 8931, pp. 80–98. Springer (2015). https://doi.org/10.1007/978-3-662-46081-8_5
21. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, Higher, Stronger: E 2.3. In: Fontaine, P. (ed.) CADE. pp. 495–507. Springer (2019)
22. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: Flanagan, C., König, B. (eds.) TACAS. pp. 407–421. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_28
23. Sutcliffe, G.: The CADE ATP System Competition - CASC. AI Magazine **37**(2), 99–101 (2016)
24. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015-2018. J. Satisf. Boolean Model. Comput. **11**(1), 221–259 (2019). https://doi.org/10.3233/SAT190123, `https://doi.org/10.3233/SAT190123`
25. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) CADE. pp. 140–145. Springer (2009). https://doi.org/10.1007/978-3-642-02959-2_10