Induction in Saturation

Laura Kovács[1(✉)], Petra Hozzová[1], Márton Hajdu[1], and Andrei Voronkov[2,3]

[1] TU Wien, Vienna, Austria
laura.kovacs@tuwien.ac.at
[2] University of Manchester, Manchester, UK
[3] EasyChair, Manchester, UK

Abstract. Proof by induction is commonplace in modern mathematics and computational logic. This paper overviews and discusses our recent results in turning saturation-based first-order theorem proving into a powerful framework for automating inductive reasoning. We formalize applications of induction as new inference rules of the saturation process, add instances of appropriate induction schemata to the search space, and use these rules and instances immediately upon their addition for the purpose of guiding induction. Our results show, for example, that many problems from formal verification and mathematical theories can now be solved completely automatically using a first-order theorem prover.

1 Introduction

Proof by induction is commonplace in modern mathematics and computational logic. Many number-theoretic arguments rely upon mathematical induction over the natural numbers, while showing correctness of software systems typically requires structural induction over inductively-defined data types, to only name two examples. The wider automation of mathematics, logic, verification and other efforts therefore demands automating induction.

Induction can be automated by reducing goals to subgoals [1,11], so that proving a goal $\forall x.F(x)$ can be proved by induction on x. However, splitting goals into subgoals and organizing proof search accordingly requires expert guidance. As an alternative, inductive reasoning has recently appeared in SMT solvers [13] and first-order theorem provers [3,12,14], complementing strong support for reasoning with theories and quantifiers. These approaches do not reduce goals to subgoals but instead implement tailored instantiations of induction schemas [3,12,14], adjust the underlying calculus with inductive generalizations [4] and function rewriting [6], extend theory reasoning for proving inductive formulas [8], and integrate induction with rewriting for generating auxiliary inductive properties during proof search [5].

This paper describes our recent efforts in these directions, entering new grounds in the automation of inductive reasoning. The distinctive feature of our work comes with mechanizing mathematical induction in saturation-based first-order theorem proving, turning thus saturation-based proof search into a powerful framework to reason about software technologies, in particular about inductive properties of functional and imperative programs.

2 Induction in Saturation - In a Nutshell

Our work combines very efficient superposition-based equational reasoning with inductive reasoning, by extending superposition with new inference rules capturing inductive steps within saturation. We refer to these inference rules as *induction rules* and consider them in addition to superposition inferences during proof-search. Following the approach of [12], we capture the application of induction via the following general induction rule:

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

where $L[t]$ is a ground literal, C is a clause, and $F \rightarrow \forall x.L[x]$ is a valid induction schema. Further, $\overline{L}[t]$ denotes the negation of $L[t]$.

In our work, we consider extensions and variants of the induction rule Ind, in order to add instances of appropriate induction schemata over inductive formulas to be proved. We call these instances *induction axioms* and automate induction in saturation via the following two, inter-connected steps:

(i) devise new induction rules;
(ii) optimize the saturation process with induction.

For step (i), we pick up a formula G in the search space and use induction rules to add new induction axioms Ax to the search space, aiming at proving $\neg G$, or sometimes a formula more general than $\neg G$. While our inference rules implement inductive reasoning upon G using Ax, adding only these inference rules to superposition-based proof search would be insufficient for efficient theorem proving. Modern saturation-based theorem provers are very powerful not just because of the logical calculi they are based on, such as superposition. What makes them powerful and efficient are redundancy criteria and pruning of the search space; strategies for directing proof search, mainly by clause and inference selection; and theory-specific reasoning, for built-in support for data types [8]. Therefore, in addition to devising new induction rules in (i), in (ii) we bring redundancy elimination, proof search options and theory axioms/rules to saturation with induction.

As a result of the combined efforts of (i)–(ii), induction in saturation maintains efficiency of standard saturation and is not limited to induction over specific (well-founded) theories. Such a genericity is particularly important for applying our results in the formal analysis of system requirements. For example, proving that every element in the computer memory is initialized or that no execution of a user request interferes with another user request, typically requires inductive reasoning with integers and arrays.

In the rest of the paper, we illustrate the automation of induction in saturation within the following three use-cases:

```
fun sum(n : ℤ) =
  if n = 0 then 0
  else n + sum(n − 1);
assert  ∀n∈ℤ.(n ≥ 0 →
        2 · sum(n) = n · (n + 1))
```

(a) Sum of the first n positive integers.

```
fun sum_evsq(n : ℤ) =
  if n = 0 then 0
  else (2 · n)^2 + sum_evsq(n − 1);
assert  ∀n∈ℤ.(n ≥ 0 →
        3·sum_evsq(n) = 2·n·(n+1)·(2·n+1))
```

(b) Sum of squares of the first n positive even integers.

Fig. 1. Inductive reasoning with integers.

– proving arithmetical properties in Sect. 3;
– enforcing safety assertions of array-manipulating programs in Sect. 4;
– reasoning about the functional correctness of programs over lists in Sect. 5.

3 Induction and Arithmetic

We first discuss our work in proving inductive properties over (sums of) integers. While integers with the standard $<$-ordering are not well-founded, we show that we can apply, and automate, induction over any integer interval with a finite bound [7].

In the sequel, we assume a distinguished *integer sort*, denoted by $\mathbb{Z}$. When we use standard integer predicates $<$, $\leq$, $>$, $\geq$, functions $+, -, \dots$ and constants $0, 1, 2, \dots$, we assume that they denote the corresponding interpreted integer predicates and functions with their standard interpretations. All other symbols are uninterpreted. We will write quantifiers like $\forall x \in \mathbb{Z}$ to denote that x has the integer sort.

Example of Induction over Integers. Consider the recursive function `sum` of Fig. 1(a), computing the sum of the integers from the integer interval $[0, n]$. We aim to prove the assertion of Fig. 1(a), denoted via **assert** and stating that the value computed by `sum` is the closed-form expression describing the sum of the first n positive integers.

In order to prove the assertion of Fig. 1(a) within saturation-based proof search, we proceed as follows. We convert the function definition of `sum` into first-order axioms and negate the assertion of Fig. 1(a), skolemizing n as σ. We obtain the following unit clauses, with each clause being implicitly universally quantified:

$$\mathtt{sum}(0) = 0 \tag{1}$$

$$n = 0 \lor \mathtt{sum}(n) = n + \mathtt{sum}(n - 1) \tag{2}$$

$$\sigma \geq 0 \tag{3}$$

$$2 \cdot \mathtt{sum}(\sigma) \neq \sigma \cdot (\sigma + 1) \tag{4}$$

Clauses (1)–(2) result from the functional definition of `sum`, whereas clauses (3)–(4) yield the clausified negation of the assertion of Fig. 1(a). We then continue by applying inference rules on these clauses with the goal of refuting the negated assertion by deriving the empty clause, corresponding to a contradiction.

Induction Rule over Integers. When considering integers, we adjust the general induction rule Ind by considering induction over well-founded (integer) intervals. In particular, for proving property (4) of Fig. 1(a), we use the following extension of the Ind rule, where b is a ground term of integer sort:

$$\frac{\overline{L}[t] \vee C \quad t \geq b}{L[b] \wedge \forall y.(y \geq b \wedge L[y] \rightarrow L[y+1]) \rightarrow \forall x.(x \geq b \rightarrow L[x])}\ (\mathsf{IntInd}_{\geq})$$

To refute the negated assertion (4), we instantiate the $\mathsf{IntInd}_{\geq}$ rule with $L[\sigma]$ being $2 \cdot \mathtt{sum}(\sigma) = \sigma \cdot (\sigma + 1)$ and b set to 0, deriving thus the following induction axiom as an instance of the induction schema of $\mathsf{IntInd}_{\geq}$:

$$\begin{aligned}&(2 \cdot \mathtt{sum}(0) = 0 \cdot (0+1)\\ &\quad \wedge\ \forall y \in \mathbb{N}.(y \geq 0 \wedge 2 \cdot \mathtt{sum}(y) = y \cdot (y+1) \implies 2 \cdot \mathtt{sum}(y+1) = (y+1) \cdot ((y+1)+1)))\\ &\implies \forall x \in \mathbb{N}.(x \geq 0 \rightarrow 2 \cdot \mathtt{sum}(x) = x \cdot (x+1))\end{aligned} \tag{5}$$

Recall that saturation-based provers work with clauses, rather than with arbitrary formulas. Therefore, the induction axiom (5) is clausified and its clausal normal form (CNF) given below is added to the search space, where y is skolemized as σ':

$$2 \cdot \mathtt{sum}(0) \neq 0 \cdot (0+1) \vee 2 \cdot \mathtt{sum}(\sigma') = \sigma' \cdot (\sigma'+1) \vee \neg(x \geq 0) \vee 2 \cdot \mathtt{sum}(x) = x \cdot (x+1) \tag{6}$$

$$\begin{aligned}&2 \cdot \mathtt{sum}(0) \neq 0 \cdot (0+1) \vee 2 \cdot \mathtt{sum}(\sigma'+1) \neq (\sigma'+1) \cdot ((\sigma'+1)+1) \vee \neg(x \geq 0) \vee\\ &\qquad\qquad 2 \cdot \mathtt{sum}(x) = x \cdot (x+1)\end{aligned} \tag{7}$$

Optimizing Induction in Saturation. Simply instantiating $\mathsf{IntInd}_{\geq}$ and adding the corresponding induction axiom for any clause $\overline{L}[t] \vee C$ in the search space would however be inefficient: considering $\overline{L}[t] \vee C$ just like any other clause in saturation may trigger the application of too many inferences. Therefore, we treat premises $\overline{L}[t] \vee C$ of induction rules differently in order to *guide the saturation algorithm* in two ways.

First, we ensure that an application of Ind or $\mathsf{IntInd}_{\geq}$ is followed by a binary resolution step in which the conclusion of an induction rule is resolved with (inductive) premise(s). For example, to derive a refutation from (6) and (7), we apply binary resolution on (6) and (7) with (3) and (4), resolving away the last two literals of (6) and (7). Refutation of (4) is then easily derived, by using the axioms (1)–(2) defining `sum` together with arithmetic reasoning over integers. For example, our theorem prover Vampire [9] finds a refutation of (4) in almost no time[1].

Second, induction can be very explosive – i.e., it may generate many consequences of which few lead to refutation. Therefore, in practice, we implement additional requirements on the premises of Ind and $\mathsf{IntInd}_{\geq}$, with these requirements to be used during saturation. Among others, we use heuristics on whether

[1] Empirical data reported in this paper have been obtained on computers with AMD Epyc 7502 2.5 GHz processors and 1 TB RAM.

```
**assume** $\text{A.size} > 0 \land \text{A}[0] = 0$

$i := 1;$
**while** $i < \text{A.size}$ **do**
   $\text{A}[i] := \text{A}[i-1] + i;$
   $i := i + 1;$
   **invariant** $\forall j \in \mathbb{Z}.(1 \leq j \leq i - 1 \rightarrow \texttt{val}_{\text{A}}(j) = \texttt{val}_{\text{A}}(j-1) + j)$
**end**

**assert** $\forall j \in \mathbb{Z}.(0 \leq j \leq \text{A.size} - 1 \rightarrow 2 \cdot \texttt{val}_{\text{A}}(j) = j \cdot (j+1))$
```

Fig. 2. Inductive reasoning with arrays, with $\texttt{val}_{\text{A}}(j)$ denoting $\text{A}[j]$.

the term t must contain a symbol from the conjecture we are trying to prove; whether we apply induction on non-unit clauses; or whether (in the case of integer induction) we allow $L[t]$ to be a comparison or equality literal, and if yes, how many times and on which positions it can contain the term t.

Induction and Theories. We note that the `sum` function and the corresponding assertion of Fig. 1(a) can also be encoded using natural numbers as inductively defined data types. While the resulting encoding of Fig. 1(a) holds over naturals, proving Fig. 1(a) over naturals becomes very complex in practice, as natural numbers do not have built-in arithmetic axioms but rely on term algebra axioms [8]. As a result, when proving Fig. 1(a) over naturals, we are faced with the challenge of proving addition and multiplication properties of naturals, which require induction as well, making efficient proof search challenging. Our work therefore advocates the combination of inductive reasoning with theory-specific inference rules, in the case of Fig. 1(a) this being the application of induction over integers.

Application of induction over integers becomes especially beneficial when proving complex, non-linear arithmetic conjectures. Figure 1(b) shows such a use-case of a function `sum_evsq` that recursively computes the sum of squares of the first n positive even integers. The assertion of Fig. 1(b) is the well-known closed form formula for the sum computed by `sum_evsq`. Proving this assertion, we follow a similar recipe as for Fig. 1(a): instantiate induction inferences over integers with the equality from the assertion, resolve the conclusion of the induction axiom with the literals of the assertion, and then prove the base case and the step case using arithmetic reasoning combined with the definition of `sum_evsq`. Thanks to theory-specific reasoning together with induction, Vampire proves Fig. 1(b) in no time (in less than 1 s).

4 Induction over Arrays

We next describe applications of induction in saturation while proving the functional correctness of array-manipulating programs.

Example of Induction over Arrays. Consider the imperative program of Fig. 2, annotated with pre-condition (**assume**), post-condition (**assert**) and loop invariant (**invariant**).

Given the pre-condition and the invariant, we aim to prove that, upon loop termination, each A$[j]$ will hold the sum of the first j positive integers. Note that the assumed termination of the loop implies the negation of the loop condition: $\neg(i < \text{A.size})$. With this additional formula, the assertion of Fig. 2 clearly holds. Yet, proving it automatically, inductive reasoning is needed.

Induction Rule over Arrays of Integers. We consider the following variant of the $\mathsf{IntInd}_{\geq}$ rule, using induction over a finite integer interval:

$$\frac{\overline{L}[t] \vee C \qquad t \geq b_1 \qquad t \leq b_2}{L[b_1] \wedge \forall x.(b_1 \leq x < b_2 \wedge L[x] \rightarrow L[x+1]) \rightarrow \forall y.(b_1 \leq y \leq b_2 \rightarrow L[y])}\ (\mathsf{IntInd}_{[\geq]}),$$

where $L[t]$ is a ground literal, and b_1, b_2 are ground terms. We instantiate $\mathsf{IntInd}_{[\geq]}$ based on the negated, skolemized and clausified assertion of Fig. 2; for doing so, we set $L[\sigma]$ to be $2 \cdot \mathtt{val}_{\text{A}}(\sigma) = \sigma \cdot (\sigma + 1)$ and consider b_1 to be 0 and b_2 to be A.size $- 1$. We further clausify the resulting induction axiom; resolve the clausified axiom against the premises of $\mathsf{IntInd}_{[\geq]}$; and finally refute the rest of the literals using the invariant, pre-condition and negated loop condition of Fig. 2 within integer arithmetic.

Note that, unlike in the examples of Fig. 1, one of the bounds of the interval upon which we are applying induction is *symbolic* – an uninterpreted constant. This is a powerful generalization which allows us to reason with arrays regardless of their specific length. In practice, induction over arrays of integers in VAMPIRE proves the assertion of Fig. 2 (using around 1 s of time).

5 Induction over Lists

We finally present our efforts towards proving inductive properties of functional programs, using combination of inductively defined data types. We use two datatypes, natural numbers and lists over natural numbers, denoted respectively by $\mathbb{N}$ and $\mathbb{L}$. We assume that these datatypes are axiomatised by the *distinctiveness*, *exhaustiveness* and *injectivity* axioms of term algebras [8].

Example of Induction over Lists. Consider the functional program of Fig. 3. We aim to prove the assertion (**assert**) expressing that reversing a list an even number of times results in the same list; doing so, we use an assumption (**assume**) corresponding to an inductive lemma. For proving the assertion of Fig. 3, we translate the function definitions and assumption of Fig. 3 into first-order axioms, negate the assertion of Fig. 3, and clausify the resulting formulas. As a result, the following two clauses are obtained from the negated assertion, respectively introducing Skolem constants σ_1 and σ_2 for n and xs:

$$\mathsf{even}(\sigma_1) \tag{8}$$

$$\mathsf{revN}(\sigma_2, \sigma_1) \neq \sigma_2 \tag{9}$$

```
fun even(n : ℕ) = match n                fun rev(xs : 𝕃) = match xs
  0 ⇒ ⊤                                    nil ⇒ nil
  s(0) ⇒ ⊥                                 cons(z, zs)) ⇒ app(rev(zs), cons(z, nil))
  s(s(m)) ⇒ even(m)

fun app(xs : 𝕃, ys : 𝕃) = match xs       fun revN(xs : 𝕃, n : ℕ) = match n
  nil ⇒ ys                                 0 ⇒ xs
  cons(z, zs) ⇒ cons(z, app(zs, ys))       s(m) ⇒ revN(rev(xs), m)

        assume ∀xs ∈ 𝕃, n ∈ ℕ.revN(xs, n) = revN(rev(rev(xs)), n)
        assert ∀n ∈ ℕ, xs ∈ 𝕃.even(n) → revN(xs, n) = xs
```

Fig. 3. Inductive reasoning with natural numbers and list datatypes.

Induction Rule over Lists. A suitable induction formula that refutes clause (9) is generated in two steps. A formula generated solely from clause (9) may be too strong. Hence, we generate an induction formula that takes clause (8) into account as well. Doing so, we use a generalization of the Ind rule that works on an arbitrary number of premises. Namely, we use the following induction rule with two premises:

$$\frac{\overline{L}[t] \vee C \quad \overline{L'}[t] \vee C'}{F \rightarrow \forall x.(L[x] \vee L'[x])} \ (\mathsf{Ind}'),$$

where $L[t]$ and $L'[t]$ are ground literals, C and C' are clauses, and $F \rightarrow \forall x.(L[x] \vee L'[x])$ is a valid induction schema.

Second, to generate a suitable antecedent for the induction schema (i.e. F), we notice that the recursion used in the definition of even suggests an induction principle different from standard structural induction over natural numbers. These insights lead us to generate following induction axiom:

$$\begin{aligned}\Big((\neg\mathsf{even}(0) \vee \mathsf{revN}(\sigma_2, 0) = \sigma_2) \ \wedge \ (\neg\mathsf{even}(\mathsf{s}(0)) \vee \mathsf{revN}(\sigma_2, \mathsf{s}(0)) = \sigma_2)\\ \wedge \ \forall n.\big((\neg\mathsf{even}(n) \vee \mathsf{revN}(\sigma_2, n) = \sigma_2) \rightarrow\\ (\neg\mathsf{even}(\mathsf{s}(\mathsf{s}(n))) \vee \mathsf{revN}(\sigma_2, \mathsf{s}(\mathsf{s}(n))) = \sigma_2)\big)\Big)\\ \rightarrow \forall m.(\neg\mathsf{even}(m) \vee \mathsf{revN}(\sigma_2, m) = \sigma_2)\end{aligned}$$

After clausifying this axiom and resolving the conclusion literals with the premises (8) and (9), a first-order refutation using the term algebra axioms and the clausified function definitions and assumption of Fig. 3 is straightforward; Vampire finds a refutation almost immediately.

Optimizing Induction in Saturation. Note that Fig. 3 uses an auxiliary inductive lemma (**assume**), in order to prove the assertion of Fig. 3. An additional challenge in automating the proof of the assertion of Fig. 3 comes therefore with the task of generating and proving auxiliary inductive lemmas during saturation.

Proving the lemma of Fig. 3 needs further induction steps; however, the generation of a suitable induction formula is only triggered by *an instance* of the respective lemma. Since the superposition calculus is optimized to avoid generating clauses unnecessary for first-order reasoning, either (i) we tweak the parameters of superposition such that the generation of an instance of the lemma *is necessary* for first-order reasoning, or (ii) we perform additional sound inferences (on top of superposition and induction inferences) to derive these instances.

Addressing these challenges, we develop different term ordering families (e.g. KBO or LPO), parameterized by various symbol precedences or weight functions; and devise literal selection functions to vary the inferred consequences of a subgoal [5]. As a result, we select different inductive lemmas during saturation. Further, we use function definitions not as axioms but as rewrite rules, in order to ensure that recursively defined functions are expanded/rewritten into their (likely much larger) definitions [6]. With such optimizations at hand, Vampire proves the assertion of Fig. 3 without using the asserted inductive lemma (**assume**), but by generating the respective inductive lemma of Fig. 3 completely automatically.

6 Conclusions and Outlook

Automated reasoning about system requirements is one of the most active areas of formal methods [2,10]. Our work addresses recent reasoning demands in the presence of induction, needed for example in proving safety and security requirements over software systems or establishing mathematical conjectures. In particular, we turn saturation-based first-order theorem proving into a powerful workhorse for automating induction. When we integrate induction in saturation, the choice of possibilities to exploit is very large. As such, should one approach fail to bring considerable improvements, one may quickly study and investigate other approaches, allowing thus for further improvements and advancements in mechanizing induction. As saturation-based first-order theorem proving is not yet fully integrated in the tech-chain of ensuring software reliability, we believe automating induction in saturation will bring significant further advances in the theory and practice of both automated reasoning and formal verification.

Acknowledgements. We thank the entire developer team of the Vampire theorem prover, and in particular Daneshvar Amrollahi, Michael Rawson, Giles Reger, and Martin Suda, for valuable discussions. We acknowledge generous funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien Doctoral College SecInt, the FWF SFB project SpyCoDe F8504, the WWTF ICT22-007 grant ForSmart, and the Amazon Research Award 2023 QuAT.

References

1. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. Academic Press, New York (1988)
2. Cook, B.: Formal reasoning about the security of Amazon web services. In: CAV, pp. 38–47 (2018)

3. Cruanes, S.: Superposition with structural induction. In: FroCoS (2017)
4. Hajdu, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: CICM (2020)
5. Hajdu, M., Kovács, L., Rawson, M.: Rewriting and inductive reasoning. In: LPAR (2024, to appear)
6. Hajdu, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: FMCAD (2021)
7. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. In: CADE, pp. 361–377 (2021)
8. Kovács, L., Robillard, S., Voronkov, A.: Coming to terms with quantified reasoning. In: POPL, pp. 260–270 (2017)
9. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: CAV (2013)
10. O'Hearn, P.W.: Continuous reasoning: scaling the impact of formal methods. In: LICS, pp. 13–25 (2018)
11. Passmore, G.O., et al.: The Imandra automated reasoning system (system description). In: IJCAR, pp. 464–471 (2020)
12. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: CADE (2019)
13. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: VMCAI, pp. 80–98 (2015)
14. Wand, D.: Superposition: types and induction. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2017). https://tel.archives-ouvertes.fr/tel-01592497