The Vampire Approach to Induction

Márton Hajdu[1], Laura Kovács[1], Michael Rawson[1] and Andrei Voronkov[2]

[1] *TU Wien*
[2] *U. Manchester and EasyChair*

Abstract

We discuss practical aspects of automating inductive reasoning in the first-order superposition prover Vampire. We explore solutions for reasoning over inductively defined dataypes; generating, storing and applying induction schema instances; and directly integrating inductive reasoning into the saturation reasoning loop of Vampire. Our techniques significantly improve the performance of Vampire despite the inherent difficulty of automated induction. We expect our exposition to be useful when implementing induction in saturation-style provers, and to stimulate further discussion and advances in the area.

Keywords

Induction, Saturation, Superposition, First-order theorem proving

1. Introduction

Due to its emerging relevance in program analysis and synthesis, inductive reasoning has seeped into major automated reasoning systems over the past few years [1, 2, 3, 4, 5, 6, 7], complementing seminal efforts in the inductive theorem proving community [8, 9]. Still, a consensus in the automated reasoning community is yet to be made on which parts of induction to automate and what to leave to the user.

In this paper we focus on *fully automatic methods to efficiently support inductive reasoning in saturation-based theorem proving*, as saturation algorithms [10] provide state-of-the-art solutions implemented by several modern first-order theorem provers [11, 12, 13]. Most provers limit their inductive capabilities to *term algebras* (also known as *inductively-defined datatypes*) [14, 15], by relying heavily on case-analysis techniques, such as the AVATAR framework [16] in [5] or duplicating the current search space in [3]. Others opt for lightweight approaches that introduce arbitrary induction formulas into the search space and let the saturation algorithm do the rest [4], thus allowing for induction on integers [17] and more general induction schemas [18]. In this paper, we describe practical aspects of integrating induction in saturation-based first-order theorem proving, in particular within the first-order theorem prover Vampire [11].

Example 1. Let us focus on the term algebras of natural numbers and lists. We consider recursive function and predicate definitions over natural numbers and lists as given in

PAAR'22: 8th Workshop on Practical Aspects of Automated Reasoning, August 11–12, 2022, Haifa, Israel
ORCID: 0000-0002-8273-2613 (M. Hajdu); 0000-0002-8299-2714 (L. Kovács); 0000-0001-7834-1567 (M. Rawson)

$$
\begin{aligned}
&\mathtt{app}(\mathtt{nil}, z) \simeq z && \text{(a1)}\\
&\mathtt{app}(\mathtt{cons}(x, y), z) \simeq \mathtt{cons}(x, \mathtt{app}(y, z)) && \text{(a2)}\\
&\mathtt{rev}(\mathtt{nil}) \simeq \mathtt{nil} && \text{(r1)}\\
&\mathtt{rev}(\mathtt{cons}(x, y)) \simeq \mathtt{app}(\mathtt{rev}(y), \mathtt{cons}(x, \mathtt{nil})) && \text{(r2)}
\end{aligned}
\qquad
\begin{aligned}
\neg\mathtt{mem}(x, \mathtt{nil}) && \text{(m1)}\\
\neg\mathtt{mem}(x, \mathtt{cons}(y, z)) \vee x \simeq y \vee \mathtt{mem}(x, z) && \text{(m2)}\\
\mathtt{mem}(x, \mathtt{cons}(y, z)) \vee x \not\simeq y && \text{(m3)}\\
\mathtt{mem}(x, \mathtt{cons}(y, z)) \vee \neg\mathtt{mem}(x, z) && \text{(m4)}
\end{aligned}
$$

Figure 1: Axiomatization of the recursive functions `app`, `rev` and the recursive predicate `mem` over the term algebra of lists; here, `nil` and `cons` denote list constructors.

Figure 1. We illustrate our work towards automating induction in saturation using the following conjecture, stating that reversing a list does not remove elements from it:

$$\forall x, y. \mathtt{mem}(x, y) \rightarrow \mathtt{mem}(x, \mathtt{rev}(y)), \tag{1}$$

where x is a natural number and y is a list. Saturation-based first-order provers establish validity of (1) by showing the unsatisfiability of its negation. Following this method, after converting the negation of (1) into clause normal form, we are left with the following two clauses:

$$\mathtt{mem}(\sigma_0, \sigma_1) \tag{2}$$

$$\neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\sigma_1)) \tag{3}$$

where σ_0, σ_1 are Skolem constants. Our approach for disproving (2)–(3), or more generally proving formulas via inductive reasoning, is to perform induction directly in saturation-based proof search, by using valid induction schemata (such as structural induction). To this end, we generate valid *induction formulas*, that is, instances of induction schemata. We further clausify these induction formulas and add them to the search space. We instantiate the structural induction schema over lists

$$\big(F[\mathtt{nil}] \wedge \forall y, z. (F[z] \rightarrow F[\mathtt{cons}(y, z)])\big) \rightarrow \forall x. F[x]$$

with the negation of clauses (2)–(3) over the term σ_1 to obtain the induction formula

$$\left(\forall y, z. \left(\begin{array}{c} \big(\neg\mathtt{mem}(\sigma_0, \mathtt{nil}) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{nil}))\big) \wedge \\ \big(\neg\mathtt{mem}(\sigma_0, z) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(z))\big) \rightarrow \\ \big(\neg\mathtt{mem}(\sigma_0, \mathtt{cons}(y, z)) \vee \mathtt{mem}(\sigma_9, \mathtt{rev}(\mathtt{cons}(y, z)))\big)\end{array}\right)\right) \rightarrow \forall x. \left(\begin{array}{c} \neg\mathtt{mem}(\sigma_0, x) \\ \vee \\ \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \end{array}\right) \tag{4}$$

We further clausify the induction formula (4) to obtain the following six clauses:

$$\mathtt{mem}(\sigma_0, \mathtt{nil}) \vee \mathtt{mem}(\sigma_0, \mathtt{cons}(\sigma_3, \sigma_2)) \vee \neg\mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \tag{4.1}$$

$$\mathtt{mem}(\sigma_0, \mathtt{nil}) \vee \neg\mathtt{mem}(\sigma_0, \sigma_2) \vee \mathtt{mem}(\sigma_1, \mathtt{rev}(\sigma_2)) \vee \neg\mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \tag{4.2}$$

$$\mathtt{mem}(\sigma_0, \mathtt{nil}) \vee \neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{cons}(\sigma_3, \sigma_2))) \vee \neg\mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \tag{4.3}$$

$$\neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{nil})) \vee \mathtt{mem}(\sigma_0, \mathtt{cons}(\sigma_3, \sigma_2)) \vee \neg\mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \tag{4.4}$$

$$\neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{nil})) \vee \neg\mathtt{mem}(\sigma_0, \sigma_2) \vee \mathtt{mem}(\sigma_1, \mathtt{rev}(\sigma_2)) \vee \neg\mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \tag{4.5}$$

$$\neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{nil})) \vee \neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{cons}(\sigma_3, \sigma_2))) \vee \neg\mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \tag{4.6}$$

We call the clauses (4.1)–(4.6), and in general clauses stemming from clausifying induction formulas, the *induction clauses* of these formulas. Notice that the non-ground literals of each clause stem from the conclusion of formula (4) — we refer to these non-ground literals as the *conclusion literals* of these clauses or of the formula (4) — while the ground literals come from the antecedent of (4). An induction formula F is *applied* on a clause set $\mathcal{C}$, or alternatively speaking $\mathcal{C}$ is *inducted upon*, when a sequence of binary resolutions is performed between the induction clauses of F and clauses of $\mathcal{C}$ such that the conclusion literals are resolved. We say we *induct on* a term t in a clause set $\mathcal{C}$ with induction formula F when, during the application of F on $\mathcal{C}$, the variable in the conclusion literals is bound to t. In Section 3, we explore solutions on how to generate and apply induction formulas in saturation.

The refutation of clauses (4.1)–(4.6) with clauses (2) and (3) goes as follows. We eliminate the conclusion literals of (4.1)–(4.6) by applying formula (4) on them, i.e. by binary resolving the six clauses first with (2), then with (3). Then, using the axioms given in Figure 1 within standard superposition rules, we obtain the following clauses:

$$\sigma_0 \simeq \sigma_3 \vee \texttt{mem}(\sigma_0, \texttt{rev}(\sigma_2)) \tag{5}$$

$$\neg\texttt{mem}(\sigma_0, \texttt{app}(\texttt{rev}(\sigma_2), \texttt{cons}(\sigma_3, \texttt{nil}))) \tag{6}$$

First, a nested induction with the same structural induction schema of lists on the term $\texttt{rev}(\sigma_2)$ in clauses (5)–(6) is performed, then using standard superposition rules we get the clause:

$$\neg\texttt{mem}(\sigma_0, \texttt{app}(\texttt{rev}(\sigma_2), \texttt{cons}(\sigma_0, \texttt{nil})))$$

One more nested induction on the term $\texttt{rev}(\sigma_2)$ in this clause results in a refutation. □

Example 1 showcases two main ingredients towards automating induction in saturation-based theorem proving: generating induction formulas/clauses and using them further in saturation. *In this paper, we address these ingredients of automating induction and present practical aspects related to applying induction formulas/clauses in saturation.* First, should we add induction clauses to the search space, at the costs of rapidly increasing our search space? If not, what inferences should we perform so that we do not lose “good" induction clauses? We present our solutions answering these two questions in Section 3. Another interesting line of work towards efficient induction in saturation comes with reusing induction formulas/clauses. To this end, a natural question arises: should we store induction formulas/clauses and query them when needed, or is it better to generate induction formulas/clauses on-demand? In Section 4 we address these questions by storing induction clauses efficiently during saturation. We also discuss more general techniques that may be integrated easily into any saturation-based theorem prover, and then focus on specialized techniques for the AVATAR architecture in Section 5. Our experiments in Section 6 showcase the practical benefits of our work.

2. Preliminaries

We briefly introduce saturation-based proof search, and refer to [11] for more details. First-order theorem provers typically work with clauses, rather than with arbitrary first-

Algorithm 1 A Simplified Saturation Loop.

```
$\mathcal{P} := \mathcal{C}, \mathcal{A} := \emptyset$
repeat
    if $\mathcal{P} = \emptyset$ then return $\mathcal{C}$ is SAT
    $C :=$ select($\mathcal{P}$)                               /* select and activate
    $\mathcal{P} := \mathcal{P} \setminus C, \mathcal{A} := \mathcal{A} \cup \{C\}$          clause $C$ from $\mathcal{P}$ */
    $\{C_1, \ldots, C_n\} :=$ derive($C, \mathcal{A}, \mathcal{I}$)           /* derive consequences of $C$
    $\mathcal{P} := \mathcal{P} \cup \{C_1, \ldots, C_n\}$                  w.r.t. $\mathcal{I}$ and $\mathcal{A}$, put them into $\mathcal{P}$ */
    if $\square \in \mathcal{P}$ then return $\mathcal{C}$ is UNSAT
```

order formulas. Given a set of assumptions $A_1, ..., A_n$ and a conjecture G, to check validity of $A_1, ..., A_1 \models G$, the set $\{A_1, ..., A_n, \neg G\}$ is transformed into an *equisatisfiable set of clauses* $\mathcal{C}$ (i.e. computing clausal normal forms via clausification). A first-order prover then *saturates* $\mathcal{C}$ by computing all logical consequences of $\mathcal{C}$ with respect to a sound inference system $\mathcal{I}$. The resulting set is called the *closure* of $\mathcal{C}$ and the process of computing the closure of $\mathcal{C}$ is called *saturation*. If the closure of $\mathcal{C}$ contains the empty clause $\square$, the original set $\mathcal{C}$ of clauses is unsatisfiable and thus G follows from $A_1, ..., A_n$. A simplified saturation algorithm for a sound inference system $\mathcal{I}$ is given in Algorithm 1, with a set of input clauses $\mathcal{C}$. We call the clause sets $\mathcal{A}$ the *active set* and $\mathcal{P}$ the *passive set*. Further, we refer to a clause from $\mathcal{A}$ as an *active clause*; similarly, clauses from $\mathcal{P}$ are *passive clauses*. One of the most common inference systems for first-order logic with equality is the *superposition calculus* [10]; many saturation provers [11, 12, 13] implement this calculus as $\mathcal{I}$ in Algorithm 1. Completeness and efficiency of saturation-based reasoning rely heavily on properties of selecting clauses from $\mathcal{P}$ and adding them to $\mathcal{A}$. A related notion is *redundancy* of clauses, which allows *simplifications* and *deletions* to eliminate redundant clauses without loss of completeness.

The AVATAR framework for first-order reasoning. The AVATAR architecture [16] is a splitting framework that combines first-order theorem proving with SAT solving. In a nutshell, the first-order prover handles proof search in AVATAR, whereas the SAT solver tracks splitting decisions and refutations at the boolean structure of clauses. An *A-clause* $C \leftarrow A$ consists of a first-order clause C and a conjunction of propositional literals (assertions) A. When AVATAR encounters a first-order clause C, it splits C into variable-disjoint *components* $C = C_1 \vee ... \vee C_n$. Then, a propositional *split clause* $[\![C_1]\!] \vee ... \vee [\![C_n]\!]$ is passed to the SAT solver, which in turn computes a model for the propositional clauses it has seen so far. If the solver reports unsatisfiability, the original problem is unsatisfiable. Otherwise, the clauses $C_i \leftarrow [\![C_i]\!]$ are added to the first-order prover. A-clauses $C \leftarrow A$ are allowed to participate in first-order proof search as long as A is satisfied by the current model. Whenever the first-order prover produces $\bot \leftarrow A$, the *conflict clause* $\neg A$ is added to the SAT solver.

3. Applying Induction Formulas in Saturation

A key ingredient in Example 1, and in general in proving inductive properties, is the generation of inductive formulas, such as (4), using an appropriate induction schema. Yet, in a saturation loop such as Algorithm 1, induction is not given special attention. All clauses are activated sequentially, and inferences are applied only over active clauses. In our approach, instances of induction schemata are first generated when a clause set matching the negation of their conclusion gets activated. This makes our induction *goal oriented*, controlling the search space explosion by only adding already applicable instances of schemata.

Example 2. We now show one possible sequence of derivation steps of Example 1 w.r.t. saturation in detail. Suppose that clause (2) is activated first. Then, we may only induct on (2), using induction formulas with a conclusion matching the negation of (2), since (3) is not yet in $\mathcal{A}$. Therefore, (4) is not generated until (3) is also activated. On the other hand, even if (3) is eventually activated and (4.1) is derived, note that (3) is not an ancestor of (4.1). However, we can *apply* (4.1) on (3) by performing the following binary resolution step:

$$\frac{\texttt{mem}(\sigma_0,\texttt{nil}) \vee \texttt{mem}(\sigma_0,\sigma_2) \vee \neg\texttt{mem}(\sigma_0,x) \vee \underline{\texttt{mem}(\sigma_0,\texttt{rev}(x))} \qquad \underline{\neg\texttt{mem}(\sigma_0,\texttt{rev}(\sigma_1))}}{\texttt{mem}(\sigma_0,\texttt{nil}) \vee \texttt{mem}(\sigma_0,\sigma_2) \vee \neg\texttt{mem}(\sigma_0,\sigma_1)} \tag{7}$$

That is, if we do not treat active clauses (4.1) and (3) as ordinary clauses but keep track of their (inductive) derivation, we could prioritize the above binary resolution step in saturation. □

Example 2 shows that the activation and use of induction formulas/clauses in saturation need a more tailored treatment. In particular, an induction inference may serve as a way of either adding instances of induction schemata to $\mathcal{P}$ or adding the application of these instances on the premises to $\mathcal{P}$. To this end, we propose the following two ways/options for *applying induction formulas in saturation*.

Addition – We allow *induction formulas to participate in any inference with any active clause*, by using induction formulas as new lemmas in proof search. In the case of Example 2, we add (4.1) to $\mathcal{P}$ without any further inference. Later, when (4.1) is activated, inference (7) with (3) may occur. After refuting the antecedent of (4), the derived clauses can be used as lemmas. Unfortunately, since these lemmas are not part of the initial search space, their use on (2) and (3) after their addition to $\mathcal{P}$ can be practically indefinitely delayed.

Resolution – We *restrict the use of induction clauses as active clauses*, and add only conclusions of binary resolutions with induction clauses to $\mathcal{P}$. In the case of Example 2, we resolve clauses (3) and (4.1), adding only the conclusion of (7) to $\mathcal{P}$. At this point, it also makes sense to perform an additional binary resolution between (2) and the conclusion of (7), and hence activate the conclusion of (7), as (2) is already in $\mathcal{A}$.

It turns out that while allowing induction formulas to interact freely as lemmas (**Addition**) blows up the search space in practice, storing and applying them in only restricted ways (**Resolution**) requires more book-keeping (Section 4). In fact, we can combine **Addition** and **Resolution**.

Example 3. Using **Addition** and **Resolution** in combination for Example 2, we add to $\mathcal{P}$ the clause (4.1) together with the result of a binary resolution between the conclusion of (7) and clause (2). Now suppose the following two clauses get activated (in any order):

$$\mathtt{mem}(\sigma_0, \mathtt{app}(\sigma_0, \sigma_1)), \tag{8}$$

$$\neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{app}(\sigma_0, \sigma_1))) \tag{9}$$

Note that these two clauses are similar to clauses (2) and (3), except they contain $\mathtt{app}(\sigma_0, \sigma_1)$ instead of σ_1. Hence, their negation matches the conclusion of induction formula (4) and we can reuse the induction formula (4) to refute also these two clauses. As (4.1) was added to saturation through $\mathcal{P}$ by **Addition**, the saturation algorithm can perform the application. However, if we want to avoid the search space explosion emitted by adding free lemmas to the search space, and only use **Resolution**, clause (4.1) does not appear in the search space and we must generate a new induction formula. To overcome the burden of generating new induction formulas that are identical to (4), we can instead store clauses (4.1)–(4.6) in an index and query these clauses when the same induction formula (4) is needed (see Section 4). □

4. Reusing Induction Formulas in Saturation

As discussed in Example 3, there may be cases where induction formulas can/must be used multiple times, but generating the same induction formulas repeatedly is inefficient. In our work, we *detect whether an induction formula has already been generated.* Additionally, for **Resolution**, we *store induction formulas to later query them for reuse.*

The obvious way to recognize, store and query clauses from induction formulas is to use term indexing [19]. The idea is to index the conclusion literals, which will be resolved with the literals inducted upon. Standard term indexing approaches are however not very efficient for induction in saturation. Many clauses of an induction formula may share conclusion literals, which — upon indexing — result in an insertion overhead, since these literals could be stored under the same entry instead of adding them separately. Moreover, we may want to avoid performing some induction inferences, for example inducting on complex terms like $\mathtt{app}(\sigma_0, \sigma_1)$ in clauses (8) and (9); yet, filtering out clauses after they have been queried from induction formulas is inefficient. When inducting on non-unit clauses, it becomes difficult to identify which induction formulas have been generated by inspecting only conclusion literals. In order to simplify indexing of induction formulas even for clause *sets*, we make the following assumptions:

Assumption 1 – *Clauses used and queried for induction are ground*, reducing the indexing problem from unification to matching.

Assumption 2 – *Induction formulas contain one induction term*, thus normalizing conclusion literals becomes easier.

With the above two assumptions, we can index a set of clauses $\{C_1, ..., C_n\}$ inducted upon with induction term t of sort τ with the following *key* function:

$$key(\{C_1, ..., C_n\}, t) := \{C_1[t/c_\tau], ..., C_n[t/c_\tau]\} \tag{10}$$

where c is a sort-indexed family of constants. A simple term index then uses the values returned by this *key* function in an associative array, in which each entry contains the induction formulas with the same conclusions, up to variable renaming.

Example 4. Using the constant $c_{\texttt{list}}$ for lists, evaluation of the *key* function with either the clause set $\{(2), (3)\}$ and induction term σ_1 or the clause set $\{(8), (9)\}$ and induction term $\texttt{app}(\sigma_0, \sigma_1)$ results in the same value:

$$key(\{(2), (3)\}, \sigma_1) = key(\{(8), (9)\}, \texttt{app}(\sigma_0, \sigma_1)) = \{\texttt{mem}(\sigma_0, c_{\texttt{list}}), \neg\texttt{mem}(\sigma_0, \texttt{rev}(c_{\texttt{list}}))\}$$

Upon querying the index with this value gives the entry with formula (4). □

Note that multiple induction formulas with the same conclusions up to variable renaming may be produced, corresponding, for example, to different well-founded relations. Hence, there may be multiple induction formulas that are mapped to the same key in the index. An interesting further direction towards weakening assumptions could come with performing unification instead of matching (**Assumption 1**) and implementing variant checking (**Assumption 2**).

We conclude by noting that the saturation loop of Algorithm 1 eagerly simplifies every generated clause C_i with respect to existing clauses before adding C_i to the passive set $\mathcal{P}$. When using induction clauses, these simplifications are performed after resolving the conclusion literals and they may be performed as many times as the induction formula is queried. In our work, we propose to *store the induction formulas already simplified* and use these simplified formulas in inferences of the saturation loop. Our experiments in Section 6 show the benefit of storing simplified induction clauses towards making proof search more efficient.

5. Improved Induction Using AVATAR

An induction clause $L_1 \vee ... \vee L_n \vee C$, with ground L_i literals and a non-ground C containing the conclusion literals, can be split into $n + 1$ components; these components are variable-disjoint as L_i are ground. Splitting the search space along $L_1 \vee ... \vee L_n \vee C$ can be done by many different splitting techniques [20], but this is done especially efficiently in Vampire using AVATAR [16].

Example 5. Continuing from the clausification of formula (4) in Example 1, let us assume that we add clauses (4.1)–(4.6) to $\mathcal{P}$ and AVATAR splits them. We name components by propositional variables and use integers to denote propositional variables. For example, we write 2 for a positive atom and $\overline{2}$ for its negation. We show below each component with its propositional variable (left), and also list the corresponding propositional split clauses (right):

$$\begin{array}{ll}
1 \leftrightarrow \llbracket \mathtt{mem}(\sigma_0, \mathtt{nil}) \rrbracket & \\
2 \leftrightarrow \llbracket \mathtt{mem}(\sigma_0, \sigma_2) \rrbracket & 1 \vee \overline{2} \vee 3 \vee 4 \\
3 \leftrightarrow \llbracket \mathtt{mem}(\sigma_0, \mathtt{rev}(\sigma_2)) \rrbracket & 1 \vee 5 \vee 4 \\
4 \rightarrow \llbracket \mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \rrbracket & 1 \vee \overline{6} \vee 4 \\
5 \leftrightarrow \llbracket \mathtt{mem}(\sigma_0, \mathtt{cons}(\sigma_3, \sigma_2)) \rrbracket & \overline{7} \vee \overline{2} \vee 3 \vee 4 \\
6 \leftrightarrow \llbracket \mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{cons}(\sigma_3, \sigma_2))) \rrbracket & \overline{7} \vee 5 \vee 4 \\
7 \leftrightarrow \llbracket \mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{nil})) \rrbracket & \overline{7} \vee \overline{6} \vee 4
\end{array}$$

The propositional split clauses are passed to the SAT solver, which reports satisfiability with a model, for example, $1 \wedge \overline{7}$, resulting in the following A-clauses added to $\mathcal{P}$:

$$\mathtt{mem}(\sigma_0, \mathtt{nil}) \leftarrow 1 \tag{11}$$

$$\neg\mathtt{mem}(\sigma_0, \mathtt{rev}(\mathtt{nil})) \leftarrow \overline{7} \tag{12}$$

Upon activation, the A-clause (11) is resolved with the axiom (m1) from Figure 1, and we derive the A-clause $\bot \leftarrow 1$, which adds the conflict clause $\overline{1}$ to the SAT solver. A new model for the updated propositional split clauses might be $\overline{1} \wedge \overline{2} \wedge 5 \wedge \overline{6}$. As shown here, the base case literal corresponding to 1 is refuted only once instead of three times in three different clauses. □

Some considerations from Sections 3 and 4 are orthogonal to splitting. When using **Addition**, saturation handles the splitting of clauses automatically. However, an interesting question is whether to store induction clauses (that is, using **Resolution**) *after splitting*. One advantage of storing induction clauses after they are split — similarly to storing them simplified — is that splitting is performed only once, not multiple times. Since only the conclusion components are used when applying an induction formula to the current premises, only these splitting components need to be stored in an index and all other components can be pushed into $\mathcal{P}$ by AVATAR when needed. In contrast, when conclusion components are queried from the index, inferences with these components should only be performed if these components are satisfied in the current SAT model, as displayed by the following example.

Example 6. Using **Resolution** after splitting clauses (4.1)–(4.6), the current model needs to be recomputed with the propositional assumption 4 (as in Example 5) to force the conclusion component. This either results in no model, in which case inferences with induction formula (4) cannot be performed and the assumption is retracted. Otherwise, a model satisfying 4 yields the following A-clause:

$$\mathtt{mem}(\sigma_0, x) \vee \mathtt{mem}(\sigma_0, \mathtt{rev}(x)) \leftarrow 4$$

Table 1
Experimental results showing solved problems with different `indfg` settings and `sic`. Parentheses indicate uniquely-solved problems within a compared group.
Legend: †: `add`, ‡: `add_resolve`, ⋆: `resolve`, ○: `regenerate`, △: `sic`

Set	Total	Strategy		†	‡	⋆	○		⋆	⋆ + △
UFDTLIA	327	1	`-indfg` comparison	143 (1)	142 (0)	171 (0)	**173** (2)	`-sic` comparison	**171** (0)	**171** (0)
		2		151 (1)	153 (0)	**190** (2)	188 (0)		**190** (0)	**190** (0)
		3		75 (0)	174 (6)	**202** (2)	200 (0)		**202** (1)	**202** (1)
		4		180 (2)	177 (0)	**216** (2)	214 (0)		216 (1)	**218** (3)
DTY	3396	1		388 (4)	385 (4)	**886** (67)	842 (32)		886 (46)	**891** (51)
		2		385 (4)	391 (11)	**878** (60)	822 (16)		**878** (46)	**878** (46)
		3		**902** (17)	899 (15)	769 (42)	722 (18)		**769** (35)	763 (29)
		4		884 (13)	**899** (19)	776 (41)	727 (16)		**776** (42)	763 (29)

After two binary resolutions with e.g. clauses (2) and (3), we obtain the empty A-clause $\bot \leftarrow 4$, yielding the conflict clause $\overline{4}$, and a SAT model for (4.1)–(4.6) is recomputed. $\square$

This overhead of recomputing SAT models in AVATAR suggests a hybrid approach: considering the size of induction clauses with and without splitting, and occasionally not splitting relatively-small clauses. An additional disadvantage of using AVATAR could be that split components are introduced even when an induction application results in only unit clauses, possibly due to simplifications. We leave further investigation of these issues as future work.

6. Implementation and Experiments

Implementation. We implemented our solutions from Sections 3–5 in Vampire[1]. Handling AVATAR split clauses turned out to be more challenging than we expected, so due to the lack of fair comparability, we do now show any results related to Section 5. We extended Vampire with the new option `induction_formula_generation` (`indfg`) to implement the techniques discussed in Section 3. The `indfg` option in Vampire has the following values: `add` for **Addition**; `resolve` for **Resolution** with the index described in Section 4; `add_resolve` for combining **Addition** and **Resolution** without an index; and `regenerate` to always generate a new induction formula. We also extended Vampire with the new option `simplify_induction_clauses` (`sic`), as in Section 4, to be used only with `resolve`.

Benchmarks. We evaluated our work using the benchmarks suite UFDTLIA from SMT-LIB [21] and our DTY inductive benchmark set [22]; both set of examples employ, among others, structural induction over term algebras.

[1] https://github.com/vprover/vampire/tree/induction-paar22

Experimental setup and results. Our experiments were performed using BenchExec [23], with a 300s time limit and 16GB memory limit, using four portfolio strategies of Vampire for structural induction. The strategies are orthogonal to the newly introduced options and are denoted with numbers 1-4. We measured how each of the `indfg` option values performed, with our results being summarized in Table 1. We also measured if enabling `sic` on top of `resolve` impacts performance. Note that with `resolve` and over 1308 proof attempts on the set UFDTLIA, the 1.29×10^8 unique induction formulas generated were applied almost 2.18×10^8 times, averaging 1.69 applications per induction formula over UFDTLIA examples which can be the reason that `resolve` has only a marginal advantage over the naive `regenerate` in these benchmarks. This number is only 1.19 for DTY, with over 1.5×10^{13} unique induction formulas, and around 1.78×10^{13} applications over 13584 proof attempts. Table 1 shows that restricted induction in saturation (**Resolution**) performs in general better (cf. legend $\star$:), while reusing and simplifying inductive formulas plays an important role (cf. `sic` option).

7. Conclusion

We implement and experimentally evaluate a number of approaches for inductive reasoning in Vampire-style saturation. We present possible ways to add consequences of induction schemata into a saturation loop: experimentally-speaking, it turns out that restricted induction **Resolution** is the best we have so far developed. Further, we show that, by careful application of indexing techniques, it is possible to reuse induction formulas to achieve higher performance, including by storing them already-simplified. Finally, we discuss the interaction of AVATAR, and clause splitting more generally, with this line of inductive reasoning.

Acknowledgements. This work was partially funded by the ERC CoG ARTIST 101002685, the EPSRC grant EP/P03408X/1, the FWF grant LogiCS W1255-N23, and the TU Wien SecInt DK.

References

[1] A. Reynolds, V. Kunčak, Induction for SMT Solvers, in: VMCAI, 2015, pp. 80–98.

[2] K. Claessen, M. Johansson, D. Rosén, N. Smallbone, HipSpec: Automating Inductive Proofs of Program Properties, in: ATx/WInG, 2012, pp. 16–25.

[3] D. Wand, Superposition: Types and Induction, Ph.D. thesis, Saarland University, 2017.

[4] G. Reger, A. Voronkov, Induction in Saturation-Based Proof Search, in: CADE, 2019, pp. 477–494.

[5] S. Cruanes, Superposition with Structural Induction, in: FroCoS, 2017, pp. 172–188.

[6] G. Passmore, S. Cruanes, D. Ignatovich, D. Aitken, M. Bray, E. Kagan, K. Kanishev, E. Maclean, N. Mometto, The Imandra Automated Reasoning System, in: IJCAR, 2020, pp. 464–471.

[7] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, CVC5: A Versatile and Industrial-Strength SMT Solver, in: TACAS, 2022, pp. 415–442.

[8] M. Kaufmann, J. S. Moore, ACL2: An Industrial Strength Version of Nqthm, in: COMPASS, 1996, pp. 23–34.

[9] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, A. Smaill, Rippling: A Heuristic for Guiding Inductive Proofs, Artif. Intell. 62 (1993) 185–253.

[10] R. Nieuwenhuis, A. Rubio, Paramodulation-Based Theorem Proving, in: Handbook of Automated Reasoning, Handbook of Automated Reasoning, 2001, pp. 371–443.

[11] L. Kovács, A. Voronkov, First-Order Theorem Proving and Vampire, in: CAV, 2013, pp. 1–35.

[12] S. Schulz, S. Cruanes, P. Vukmirovic, Faster, Higher, Stronger: E 2.3, in: CADE, 2019, pp. 495–507.

[13] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, SPASS version 3.5, in: CADE, 2009, pp. 140–145.

[14] L. Kovács, S. Robillard, A. Voronkov, Coming to Terms with Quantified Reasoning, in: POPL, 2017, pp. 260–270.

[15] J. C. Blanchette, N. Peltier, S. Robillard, Superposition with Datatypes and Codatatypes, in: IJCAR, 2018, pp. 370–387.

[16] A. Voronkov, AVATAR: The Architecture for First-Order Theorem Provers, in: CAV, 2014, pp. 696–710.

[17] P. Hozzová, L. Kovács, A. Voronkov, Integer Induction in Saturation, in: CADE, 2021, pp. 361–377.

[18] M. Hajdú, P. Hozzová, L. Kovács, A. Voronkov, Induction with Recursive Definitions in Superposition, in: FMCAD, 2021, pp. 1–10.

[19] R. Sekar, I. Ramakrishnan, A. Voronkov, Term Indexing, in: Handbook of Automated Reasoning, Handbook of Automated Reasoning, 2001, pp. 1853–1964.

[20] K. Hoder, A. Voronkov, The 481 Ways to Split a Clause and Deal with Propositional Variables, in: CADE, volume 7898, 2013, pp. 450–464.

[21] C. Barrett, P. Fontaine, C. Tinelli, The Satisfiability Modulo Theories Library (SMT-LIB), `www.SMT-LIB.org`, 2016.

[22] M. Hajdú, P. Hozzová, L. Kovács, J. Schoisswohl, A. Voronkov, Inductive Benchmarks for Automated Reasoning, in: F. Kamareddine, C. S. Coen (Eds.), CICM, 2021, pp. 124–129.

[23] D. Beyer, Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016), in: TACAS, 2016, pp. 887–904.