EasyChair Preprint

№ 826

Induction in Saturation-Based Proof Search

Giles Reger and Andrei Voronkov

March 12, 2019

Induction in Saturation-Based Proof Search *

Giles Reger[1] and Andrei Voronkov[1,2]

[1] University of Manchester, Manchester, UK
[2] EasyChair

Abstract. Many applications of theorem proving, for example program verification and analysis, require first-order reasoning with both quantifiers and theories such as arithmetic and datatypes. There is no complete procedure for reasoning in such theories but the state-of-the-art in automated theorem proving is still able to reason effectively with real-world problems from this rich domain. In this paper we introduce a missing part of the puzzle: automated induction inside a saturation-based theorem prover. Our goal is to incorporate lightweight automated induction in a way that complements the saturation-based approach, allowing us to solve problems requiring a combination of first-order reasoning, theory reasoning, and inductive reasoning. We implement a number of techniques and heuristics and evaluate them within the Vampire theorem prover. Our results show that these new techniques enjoy practical success on real-world problems.

1 Introduction

Saturation-based proof search has been the leading technology in automated theorem proving for first-order logic for some time. The core idea of this approach is to saturate a set of clauses (including the negated goal) with respect to some inference system with the aim of deriving a contradiction and concluding that the goal holds. Over the last few years this technology has been extended to reason with both quantifiers, and theories such as arithmetic and term algebras (also known as algebraic, recursive or inductive datatypes), making it highly applicable in areas such as program analysis and verification, which were previously the sole domain of SMT solvers. However, so far little has been done to extend saturation-based proof search with *automated induction*. Most attempts to date have focussed on using saturation-based methods to discharge subgoals once an induction axiom has been selected.

The aim of this work is to extend saturation-based proof search with *lightweight* methods for automated induction where those techniques are integrated directly into proof search i.e. they do not rely on some external procedure to produce subgoals. We achieve this by the introduction of new inference rules capturing inductive steps and new proof search heuristics to guide their application.

* This work was supported by EPSRC Grant EP/P03408X/1. Andrei Voronkov was also partially supported by ERC Starting Grant 2014 SYMCAR 639270 and the Wallenberg Academy Fellowship 2014 – TheProSE.

Our approach is based on the research hypothesis that many problems requiring induction only require relatively simple applications of induction.

Example 1. As an introductory example, consider the problem of proving the commutativity of $(\forall x \forall y)\mathsf{plus}(x, y) \approx \mathsf{plus}(y, x)$, where x and y range over natural numbers. We now briefly described how this approach will handle this problem.

When we Skolemise its negation, we obtain the clause $\mathsf{plus}(\sigma_0, \sigma_1) \not\approx \mathsf{plus}(\sigma_1, \sigma_0)$. In this paper, we will denote by σ_i fresh Skolem constants introduced by converting formulas to clausal form.

Our approach will immediately apply induction to σ_0 in the negated conjecture by resolving this clause with the (clausal form of the) induction axiom

$$\left(\begin{array}{l}\mathsf{plus}(\mathsf{zero}, \sigma_1) \approx \mathsf{plus}(\sigma_1, \mathsf{zero}) \wedge \\ (\forall z)\left(\begin{array}{l}\mathsf{plus}(z, \sigma_1) \approx \mathsf{plus}(\sigma_1, z) \rightarrow \\ \mathsf{plus}(\mathsf{succ}(z), \sigma_1) \approx \mathsf{plus}(\sigma_1, \mathsf{succ}(z))\end{array}\right)\end{array}\right) \rightarrow (\forall x)\mathsf{plus}(x, \sigma_1) \approx \mathsf{plus}(\sigma_1, x)$$

to produce the following subgoals:

$$\begin{array}{l}\mathsf{plus}(\mathsf{zero}, \sigma_1) \not\approx \mathsf{plus}(\sigma_1, \mathsf{zero}) \vee \mathsf{plus}(\mathsf{succ}(\sigma_2), \sigma_1) \not\approx \mathsf{plus}(\sigma_1, \mathsf{succ}(\sigma_2)) \\ \mathsf{plus}(\mathsf{zero}, \sigma_1) \not\approx \mathsf{plus}(\sigma_1, \mathsf{zero}) \vee \mathsf{plus}(\sigma_1, \sigma_2) \approx \mathsf{plus}(\sigma_2, \sigma_1)\end{array} \tag{1}$$

Clause splitting is then used to split the search space into two parts to be considered separately. This splitting is important to our approach and can be used in any saturation theorem prover implementing some version of it, for example using splitting with backtracking as in SPASS [23] or the AVATAR architecture as in Vampire [21]. The first part contains $\mathsf{plus}(\mathsf{zero}, \sigma_1) \not\approx \mathsf{plus}(\sigma_1, \mathsf{zero})$ and is refuted by deriving $\mathsf{plus}(\sigma_1, \mathsf{zero}) \not\approx \sigma_1$ using the definition of plus and applying a second induction step to σ_1 in this clause. By resolving with a similar induction axiom to before, the following clauses are produced and are refuted via the definition of plus and the injectivity of datatype constructors.

While inductive reasoning in this example may seem to be the same as in almost any other inductive theorem prover, there is an essential difference: instead of reducing goals to subgoals using induction and trying to prove these subgoals using theory reasoning or again induction, we simply consider induction as an additional inference rule adding new formulas to the search space. In a way, every clause generated during the proof search becomes a potential target for applying induction and induction becomes integrated in the saturation process.

$$\begin{array}{l}\mathsf{zero} \not\approx \mathsf{plus}(\mathsf{zero}, \mathsf{zero}) \vee \mathsf{succ}(\sigma_3) \not\approx \mathsf{plus}(\mathsf{succ}(\sigma_3), \mathsf{zero}) \\ \mathsf{zero} \not\approx \mathsf{plus}(\mathsf{zero}, \mathsf{zero}) \vee \mathsf{plus}(\sigma_3, \mathsf{zero}) \approx \sigma_3\end{array}$$

The second part of the clause splitting then contains the other half of the clauses given above. Superposition is then applied to these clauses and the axioms of plus to derive

$$\mathsf{succ}(\mathsf{plus}(\sigma_1, \sigma_2)) \not\approx \mathsf{plus}(\sigma_1, \mathsf{succ}(\sigma_2))$$

and a third induction step is applied to this clause on σ_1. The resulting subgoals can again be refuted via the definition of plus and the injectivity of datatype constructors.

In this example there were three applications of induction to ground unit clauses in the search space, however our implementation performs 5 induction steps with 2 being unnecessary for the proof. This is typical in saturation-based proof search where many irrelevant consequences are often derived. This is an important observation; our general approach is to derive consequences (inductive or otherwise) in a semi-guided fashion, meaning that we may make many unnecessary induction steps. However, this is the philosophy behind saturation-based approaches.

During proof search for this example it was necessary to (i) decide which clauses to apply induction to, (ii) decide which term within that clause to apply induction to, and (iii) decide how to apply induction. We address issues (ii) and (iii) in this paper, whilst relying on the clause selection techniques of saturation-based theorem provers for (i). We begin in Section 2 by introducing the necessary preliminary definitions for the work. In Section 3 we address (iii), how we apply induction, through the introduction of a set of new inference rules. In Section 4 we consider (ii) through a number of heuristics for selecting goals for induction. Then in Section 5 we show how standard clause splitting techniques can be used in our induction proofs (without any additional work) for case splitting. Section 6 describes implementation and experimental evaluation. We then consider related work in Section 7 before concluding in Section 8.

2 Preliminaries

Multi-Sorted First-Order Logic. We consider standard multi-sorted first-order predicate logic with equality. We allow all standard boolean connectives and quantifiers in the language. Throughout this paper, we denote terms by s, t, variables by x, y, z, constants by a, and function symbols by f, all possibly with indices. We consider equality $\approx$ as part of the language, that is, equality is not a symbol. For simplicity, we write $s \not\approx t$ for the formula $\neg(s \approx t)$.

An *atom* is an equality or a predicate applied to a list of terms. A *literal* is an atom A or its negation $\neg A$. Literals that are atoms are called *positive*, while literals of the form $\neg A$ are *negative*. If $L = \neg A$ is a literal we write $\neg L$ for the literal A. A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$, where $n \geq 0$. When $n = 0$, we will speak of the empty clause, denoted by $\Box$. We denote atoms by A, literals by L, clauses by C, and formulas by F, all possibly with indices.

By an *expression* E we mean a term, atom, literal, or clause. We write $E[t]$ to mean an expression E with a particular occurrence of a term t and then $E[s]$ for that expression with the particular occurrence of t replaced by term s.

The free variables of a formula are those not bound by a quantifier. Let F be a formula with free variables $\bar{x}$, then $\forall F$ denotes the formula $(\forall \bar{x})F$. A formula is called *closed* if it has no free variables. A formula, literal or term is called *ground* if it has no occurrences of variables. Closed formulas can be *clausified* (transformed into a set of clauses) via standard techniques (e.g. [12] and our recent work in [14]). We write $\mathsf{clausify}(F)$ for the set of clauses obtained from F by clausification.

A *multi-sorted signature* is a finite set of symbols and a finite set of sorts with the accompanying function *srt* providing sorts for the symbols.

The Theory of Finite Term Algebras. In this paper we consider induction for finite term algebras, also known as as algebraic, inductive, or recursive datatypes. A definition of the first-order theory of term algebras over a finite signature can be found in e.g. [16] and a description of how saturation-based proof search may be extended to reason with such structures is given in [8].

Let Σ be a finite set of function symbols containing at least one constant. Denote by $\mathcal{T}(\Sigma)$ the set of all ground terms built from the symbols in Σ. The *Σ-term algebra* is the algebraic structure whose carrier set is $\mathcal{T}(\Sigma)$ and defined in such a way that every ground term is interpreted by itself (we leave details to the reader).

We will often consider extensions of term algebras by additional symbols. Elements of Σ will be called *term constructors* (or simply just *constructors*), to distinguish them from other function symbols. We will differentiate between *recursive* constructors that are recursive in their arguments and *base* constructors that are not. Where we wish to differentiate we may write $\mathcal{T}(\Sigma_B, \Sigma_R)$ for base constructors Σ_B and recursive constructors Σ_R.

In practice, it can be useful to consider multiple sorts, especially for problems taken from functional programming. In this setting, each term algebra constructor has a type $\tau_1 \times \cdots \times \tau_n \to \tau$. The requirement that there is at least one constant should then be replaced by the requirement that for every sort, there exists a ground term of this sort. We also consider theories, which mix constructor and non-constructor sorts. That is, some sorts contain constructors and some do not (e.g. arithmetic).

Finally, we associate n *destructor* (or projection) functions with every constructor c of arity n such that each destructor returns one of the arguments of c. Note, that the behavior of a destructors are unspecified on some terms.

Example 2. We introduce two term algebras. Firstly, that of natural numbers

$$nat := \mathsf{zero} \mid \mathsf{succ}(\mathsf{dec}(nat))$$

and secondly that of integer lists

$$list := \mathsf{nil} \mid \mathsf{cons}(\mathsf{hd}(Int), \mathsf{tail}(list))$$

note that this second term algebra relies on an inbuilt *Integer* sort.

Saturation-Based Proof Search. An important concept in this work is that of saturation with respect to an inference system. Inference systems are used in the theory of superposition [11] implemented by several leading automated first-order theorem provers, including Vampire [9] and E [17]. Superposition theorem provers implement proof-search algorithms in $\mathcal{S}$ using so-called *saturation algorithms*, as follows. Given a set S of formulas, superposition-based theorem provers try to saturate S with respect to $\mathcal{S}$, that is build a set of formulas that

```
input: Init: set of clauses;
var active, passive, unprocessed: set of clauses; var given, new: clause;
active := ∅; unprocessed := Init;
loop
  while unprocessed ≠ ∅
    new:=pop(unprocessed);
    if new = □ then return unsatisfiable;
    add new to passive
  if passive = ∅ then return satisfiable or unknown
  given := select(passive);                          (* clause selection *)
  move given from passive to active;
  unprocessed:=infer(given, active);                 (* generating inferences *)
```

Fig. 1. Simple Saturation Algorithm.

contains S and is closed under inferences in $\mathcal{S}$. At every step, a saturation algorithm selects an inference of $\mathcal{S}$, applies this inference to S, and adds conclusions of the inferences to the set S. If at some moment the empty clause $\Box$ is obtained, by soundness of $\mathcal{S}$, we can conclude that the input set of clauses is unsatisfiable. Figure 1 gives a simple saturation algorithm. This algorithm is missing an important notion of *redundancy*. We have omitted this as it does not interact with the elements of proof search we consider here. However, it is core to the implementation in the Vampire theorem prover. It is important to note that the only way to guide proof search is via how we select clauses and how we perform inferences on them.

3 Performing Induction

In this section we introduce inference rules for performing induction for term algebras.

3.1 General Approach We begin by describing our general approach. The idea is to add inference rules that capture the application of induction to the selected clause in proof search. These inference rules will be applied during proof search to selected clauses in the same way as other inference rules such as resolution. We define an *induction axiom* to be any valid (in the underlyng theory) formula of the form

$$\mathit{formula} \rightarrow (\forall x)(L[x]).$$

For simplicity we assume that this formula is closed, leaving out the general case due to the lack of space. The idea is to *resolve* this with a clause $\neg L[t] \vee C$ obtaining $\mathit{formula} \rightarrow C$. Again, for simplicity we assume that t is a ground term. As long as the induction axiom is valid, this approach is always *sound*. If the resulting formula is not a clause, it should then be converted to its CNF.

The idea is that $L[t]$ is a *(sub)goal* we are trying to prove. This is an interesting point. Typically, saturation-based proof search is not goal-oriented (although one can introduce heuristics that support this) but this approach to induction is goal-oriented in nature as the conclusion of an induction inference is a subgoals

that, if refuted, proves the goal represented by the premise. Also, similar to [6] by resolving the induction axiom to reduce the goal to subgoals we bypass the literal selection used in saturation algorithms. This means that, if we would just add the (clausal form of) the induction axiom to the search space, we would most likely never use it to resolve against the goal in the same way as above since the literal $L[x]$ would not necessarily be selected.

Below we consider two different kinds of induction axioms, introducing three inference rules, parametrised by some term algebra. To formalise the selection of goals that can be proved by induction we introduce a predicate $sel(C, L, t)$ that is true if C is clause, L a literal in C and t a term in L. We will call this predicate *induction heuristics* since it will be used to decide when induction should be applied. In this case we will informally say that t is the induction term and L the induction literal in C.

We will first concentrate on how induction should be performed once an induction term and literal have been selected, and then discuss various choices for selection.

3.2 Structural Induction We begin by motivating the inference rule by the simple example of inductively proving that the length of a list is non-negative.

Example 3 (Structural Induction on Lists). Consider the following conjecture $(\forall x : \mathit{list})(\mathsf{len}(x) \geq 0)$ for integer lists (defined in Example 2) given the axioms $\mathsf{len}(\mathsf{nil}) \approx 0$ and $(\forall x : \mathit{Int}, y : \mathit{list})(\mathsf{len}(\mathsf{cons}(x, y)) \approx 1 + \mathsf{len}(y))$ for the len function. To prove this conjecture we must first negate it to get $\neg(\mathsf{len}(\sigma) \geq 0)$ and then introduce the induction axiom

$$(\mathsf{len}(\mathsf{nil}) \geq 0 \wedge (\forall x, y : \mathsf{len}(x) \geq 0 \rightarrow \mathsf{len}(\mathsf{cons}(y, x)) \geq 0)) \rightarrow (\forall x)(\mathsf{len}(x) \geq 0)$$

which is then resolved against $\neg(\mathsf{len}(\sigma) \geq 0)$ to give, after conversion to CNF, two clauses

$$\neg(\mathsf{len}(\mathsf{nil}) \geq 0) \vee \mathsf{len}(\sigma_1) \geq 0$$
$$\neg(\mathsf{len}(\mathsf{nil}) \geq 0) \vee \neg(\mathsf{len}(\mathsf{cons}(\sigma_2, \sigma_1)) \geq 0),$$

which can be refuted using the axioms for len. The question now is what inference rule is needed for performing the above induction step. To do so, we define the induction heuristics $sel(C, L, t)$ to hold when L is the only literal in C and t is a constant of the sort *list*. This rule effectively results in the following inferences performed by a saturation theorem prover:

$$\frac{\neg A[a]}{\begin{array}{l}\neg A[\mathsf{nil}] \vee A[\sigma_1] \\ \neg A[\mathsf{nil}] \vee \neg A[\mathsf{cons}(\sigma_2, \sigma_1)]\end{array}}$$

where a is a constant, $L[a]$ is ground, $srt(a) = \mathsf{list}$ and σ_1, σ_2 are fresh constants.

3.3 Well-Founded Induction Suppose that $x \succ y$ is any binary predicate that is interpreted as a well-founded relation (which is not necessarily an ordering). We require both arguments of $\succ$ to be of the same sort. Then the following is a valid formula, which represents well-founded induction on this relation:

$$\forall x(\neg L[x] \rightarrow \exists y(x \succ y \wedge \neg L[y])) \rightarrow \forall x L[x].$$

When we skolemize this formula, we obtain two clauses

$$\neg L[\sigma_1] \vee L[x]$$
$$\neg \sigma_1 \succ y \vee \neg L[y] \vee L[x]$$

We can use the following two equivalent clauses instead:

$$\begin{array}{l} \neg L[\sigma_1] \\ \neg \sigma_1 \succ y \vee \neg L[y] \vee L[x] \end{array} \tag{2}$$

Well-founded induction is the most general form of induction (though in practice it can only be used when the relation $\succ$ can be expressed in the first-order language we are using). We are interested in finding special cases of well-founded induction for term algebras. There are two obvious candidates for it: the immediate subterm relation $\succ_1$ and the subterm relation considered below.

Let us begin with the immediate subterm relation. Note that the relation $\succ$ must have both arguments of the same sort, so the corresponding induction rule will only be useful for term algebras where at least one argument of a constructor has the same sort as the constructor itself. Fortunately, this is the case for the three most commonly used inductive data types: natural numbers, lists and trees.

Let us provide a complete axiomatisation of the immediate subterm relation $\succ_i$ first for natural numbers and lists:

$$\neg \mathsf{zero} \succ_1 x \qquad \neg \mathsf{nil} \succ_1 x$$
$$\mathsf{succ}(x) \succ_1 y \leftrightarrow x \approx y \qquad \mathsf{cons}(x, y) \succ_1 z \leftrightarrow y \approx z$$

The subterm relation is generally not axiomatisable. However, this is not a problem in general, since we can use as an incomplete axiomatisation of the subterm relation any set of formulas which are true on this relation (though this restricts what can be proved about the relation). If we then prove anything using this set of formulas, then our proof will be correct for the subterm relation too, which makes the corresponding induction rule valid too.

We can generalise the immediate subterm and subterm relation also to trees and some other (but not all!) inductively defined types. We do not include general definitions here as they become very involved with multiple sorts and mutually recursive type definitions.

3.4 Inductive Strengthening We now consider a different form of induction axiom (inspired by [15]).

Example 4. Given the negated conjecture $\neg(\mathsf{len}(\sigma_1) \geq 0)$ given in Example 3 we consider a different way in which to inductively demonstrate $L[x]$ and thus refute this claim. The idea here is to argue that if there does not exist a *smallest* list of non-negative length then the length of all lists is non-negative. This can be captured in the induction axiom

$$\neg(\exists x) \left(\begin{array}{l} \neg(\mathsf{len}(x) \geq 0) \wedge \\ (\forall y)(\mathsf{subterm}_{list}(x, y) \rightarrow \mathsf{len}(\mathsf{tail}(y)) \geq 0) \end{array} \right) \rightarrow (\forall z)(\mathsf{len}(z) \geq 0)$$

where $\mathsf{subterm}_{list}(x, y)$ is true if y is a subterm of x of *list* sort. However, as argued in [8], the $\mathsf{subterm}$ relation needs to be axiomatised and these axioms (which include transitivity) can have a large negative impact on the search space. Therefore, we can consider two alternative inductive axioms. The first is the *weak* form where we consider only *direct* subterms of x as follows.

$$\neg(\exists x)\begin{pmatrix}\neg(\mathsf{len}(x) \geq 0) \wedge \\ (x \approx \mathsf{cons}(\mathsf{hd}(x), \mathsf{tail}(x)) \rightarrow \mathsf{len}(\mathsf{tail}(x)) \geq 0)\end{pmatrix} \rightarrow (\forall y)(\mathsf{len}(y) \geq 0)$$

This is clausified as

$$\begin{array}{l}\mathsf{len}(x) \geq 0 \vee \neg(\mathsf{len}(\sigma_2) \geq 0) \\ \mathsf{len}(x) \geq 0 \vee \sigma_2 \not\approx \mathsf{cons}(\mathsf{hd}(\sigma_1), \mathsf{tail}(\sigma_2)) \vee \mathsf{len}(\mathsf{tail}(\sigma_2)) \geq 0\end{array}$$

which can be resolved against $\neg(\mathsf{len}(\sigma_1) \geq 0)$ as before.

The second (taken from [8]) is where we represent the subterm relation in a way that is more friendly to saturation-based theorem provers i.e. we introduce a fresh predicate less and then add axioms such that it holds for exactly those terms smaller than the existential witness x. This can be written as follows.

$$\neg(\exists x)\begin{pmatrix}(x \approx \mathsf{cons}(\mathsf{hd}(x), \mathsf{tail}(x)) \rightarrow \mathsf{less}(\mathsf{tail}(x))) \wedge \\ (\forall y)(\mathsf{less}(\mathsf{cons}(\mathsf{hd}(y), \mathsf{tail}(y))) \rightarrow \mathsf{less}(\mathsf{tail}(y))) \wedge \\ (\forall z)(\mathsf{less}(z) \rightarrow \neg(\mathsf{len}(z) \geq 0))\end{pmatrix} \rightarrow (\forall y)(\mathsf{len}(y) \geq 0)$$

Again, the specific approach taken in this example can be generalised to the arbitrary term algebra $ta = \mathcal{T}(\Sigma_B \cup \Sigma_R)$. The existential part $exists_{ta}(L)$ of the general induction axiom can be given as

$$(\exists x)\left(\neg L[x] \bigwedge_{\mathsf{con}(\ldots,\mathsf{d_i},\ldots) \in \Sigma_R} (x \approx \mathsf{con}(\ldots, \mathsf{d}_i(x), \ldots) \rightarrow L[x])\right)$$

for the first approach and as

$$(\exists x)\begin{pmatrix}\bigwedge_{\mathsf{con}(\ldots,\mathsf{d_i},\ldots) \in \Sigma_R} \bigwedge_{j \in rec(\mathsf{con})} x \approx \mathsf{con}(\ldots, \mathsf{d}_i(x), \ldots) \rightarrow \mathsf{less}(\mathsf{d_j}(x)) \wedge \\ (\forall y)(\bigwedge_{\mathsf{con}(\ldots,\mathsf{d_i},\ldots) \in \Sigma_R} \bigwedge_{j \in rec(\mathsf{con})} \mathsf{less}(\mathsf{con}(\ldots, \mathsf{d}_i(x), \ldots)) \rightarrow \mathsf{less}(\mathsf{d}_j(y))) \wedge \\ (\forall z)(\mathsf{less}(z) \rightarrow \neg L[z])\end{pmatrix}$$

for the second approach. The general induction rule then becomes

$$\frac{L[t] \vee C}{\mathsf{clausify}(exists_{ta}(\neg L) \vee C)}$$

for ground literal $L[t]$, clause C and term t, where $srt(t) = ta$ and $sel(L[t] \vee C, L[t], t)$.

One could consider an optimisation where this approach is applied directly to the input (as is done in [15]). However, this would introduce induction axioms too early in proof search i.e. it goes against the saturation-based philosophy. One could also consider reusing Skolem constants instead of introducing new ones where t in the above rule is already a Skolem constant. However, this could only be done for each Skolem constant at most once.

Approach One

$$\frac{L[t] \vee C}{\begin{array}{l} C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee \neg L[\sigma_4] \vee \neg L[\sigma_2] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee \neg L[\sigma_4] \vee \neg L[\sigma_6] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee \neg L[\sigma_7] \vee \neg L[\sigma_2] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee \neg L[\sigma_7] \vee \neg L[\sigma_6] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee L[\mathsf{black}(\sigma_3, \sigma_7, \sigma_4)] \vee \neg L[\sigma_2] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee L[\mathsf{black}(\sigma_3, \sigma_7, \sigma_4)] \vee \neg L[\sigma_6] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee L[\mathsf{red}(\sigma_5, \sigma_2, \sigma_6)]] \vee \neg L[\sigma_7] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee L[\mathsf{red}(\sigma_5, \sigma_2, \sigma_6)] \vee \neg L[\sigma_4] \\ C \vee L[\mathsf{empty}] \vee L[\mathsf{leaf}(\sigma_1)] \vee L[\mathsf{black}(\sigma_3, \sigma_7, \sigma_4)] \vee L[\mathsf{red}(\sigma_5, \sigma_2, \sigma_6)] \end{array}}$$

Approach Two

$$\frac{L[t] \vee C}{\begin{array}{l} C \vee \mathsf{red}(\mathsf{rval}(\sigma_1), \mathsf{rleft}(\sigma_1), \mathsf{rright}(\sigma_1)) \not\approx \sigma_1 \vee \neg L[\mathsf{rleft}(\sigma_1)] \\ C \vee \mathsf{red}(\mathsf{rval}(\sigma_1), \mathsf{rleft}(\sigma_1), \mathsf{rright}(\sigma_1)) \not\approx \sigma_1 \vee \neg L[\mathsf{rright}(\sigma_1)] \\ C \vee \mathsf{black}(\mathsf{bval}(\sigma_1), \mathsf{bleft}(\sigma_1), \mathsf{bright}(\sigma_1)) \not\approx \sigma_1 \vee \neg L[\mathsf{bleft}(\sigma_1)] \\ C \vee \mathsf{black}(\mathsf{bval}(\sigma_1), \mathsf{bleft}(\sigma_1), \mathsf{bright}(\sigma_1)) \not\approx \sigma_1 \vee \neg L[\mathsf{bright}(\sigma_1)] \\ C \vee L[\sigma_1] \end{array}}$$

Approach Three

$$\frac{L[t] \vee C}{\begin{array}{l} C \vee L[\sigma_1] \\ C \vee \mathsf{red}(\mathsf{rval}(\sigma_1), \mathsf{rleft}(\sigma_1), \mathsf{rright}(\sigma_1) \not\approx \sigma_1 \vee \mathsf{less}(\mathsf{rleft}(\sigma_1)) \\ C \vee \mathsf{red}(\mathsf{rval}(\sigma_1), \mathsf{rleft}(\sigma_1), \mathsf{rright}(\sigma_1) \not\approx \sigma_1 \vee \mathsf{less}(\mathsf{rright}(\sigma_1)) \\ C \vee \mathsf{black}(\mathsf{bval}(\sigma_1), \mathsf{bleft}(\sigma_1), \mathsf{bright}(\sigma_1) \not\approx \sigma_1 \vee \mathsf{less}(\mathsf{bleft}(\sigma_1)) \\ C \vee \mathsf{black}(\mathsf{bval}(\sigma_1), \mathsf{bleft}(\sigma_1), \mathsf{bright}(\sigma_1) \not\approx \sigma_1 \vee \mathsf{less}(\mathsf{bright}(\sigma_1)) \\ C \vee \neg\mathsf{less}(x) \vee \neg L[x] \end{array}}$$

Table 1. Illustrating instantiating the induction inference schemas for the *rbtree* term algebra.

3.5 Comparing the Approaches with an Example To illustrate the differences in the clauses produced by the above three approaches we instantiate each inference rule schema by a more complex term algebra defined as

$$\begin{array}{lll} \mathit{rbtree} := \mathsf{empty} & \mid \mathsf{leaf}(\mathsf{lval}(\mathit{Int})) & \mid \mathsf{red}(\mathsf{rval}(\mathit{Int}, \mathsf{rleft}(\mathit{rbtree}), \mathsf{rright}(\mathit{rbtree})) \\ & & \mid \mathsf{black}(\mathsf{bval}(\mathit{Int}), \mathsf{bleft}(\mathit{rbtree}), \mathsf{bright}(\mathit{rbtree})) \end{array}$$

This covers all the important cases from above (i) non-zero arity base constructors, and (ii) multiple base and multiple recursive constructors. Table 1 gives the introduced inference rules instantiated with $ta = \mathit{rbtree}$. Notice how the structural induction rule, in this case introduces 7 new Skolem constants and 9 clauses (although this could be slightly optimised here) whilst the inductive strengthening approaches introduce one Skolem constant and fewer clauses.

4 Selecting Where to Apply Induction

We now consider how to define various induction heuristics.

4.1 Goal-Directed Search If we consider our introductory example (Example 1) of proving the commutativity of addition then we observe that we (usefully) applied induction three times. The first time was directly to the goal and the second two times were to unit clauses derived directly from the result of this first induction. We hypothesise that this is a typical scenario and introduce heuristics that represents for this common case. An important observation here is that an implicative universal goal becomes a set of unit ground clauses once negated.

Unit clauses. A unit clause represents a single goal or subgoal that, if refuted, will lead to a final proof. Conversely, applying the above induction inference rules to non-unit clauses will lead to applications of induction that may not be as general as needed. This selection can be defined as follows for some literal $L[t]$ and term t.

$$sel_U(L[t], L[t], t)$$

Negative literals. Typically, goal statements are positive and therefore proof search is attempting to derive a contradiction from a negative statement. Applying induction to a negative statement leads to a mixture of positive and negative conclusions. As we saw in the introductory example, it is common to apply further induction to the negative conclusions. This selection can be defined as follows for clause C, atom A and term t.

$$sel_U(C \vee \neg A[t], \neg A[t], t)$$

However, it is easy to see cases where this is too restrictive. For example, the goal from Example 3 could have been rewritten as $(\forall x)(\neg(\mathsf{len}(x) < 0))$ and the negated goal on which induction should be performed would have been positive.

Constants. Given a purely universal goal, the terms of interest will be Skolem constants (whether this Skolemisation occurred within the solver or not) and terms introduced by induction for repeated induction or also typically Skolem constants. Therefore, to restrict application of induction to this special case, we can restrict it to constants only. This selection can be defined as follows for clause C, literal L and constant a.

$$sel_C(C \vee L[a], L[a], a)$$

Special symbols. The goal will typically contain the symbols on which induction should be performed. Additionally, further induction steps are often performed on the Skolem constants introduced by a previous induction. We define a selection predicate parameterised by a set of symbols α as follows for clause C, literal L and term t.

$$sel_\alpha(C \vee L[t], L[t], t) \Leftrightarrow (t = f(t_1, \ldots, t_n) \rightarrow f \in \alpha) \wedge (t = a \rightarrow a \in \alpha)$$

and define the functions sel_G and sel_I for sets of goal symbols G and induction Skolem constants G.

The *sel* function is then defined as any *conjunction* of zero or more of the above with the trivial selection function that is true on all inputs where t is of term algebra sort.

4.2 Inferring Goal Clause(s) One issue with the above heuristics is that we may not have an explicit goal in our input problem. Indeed, SMT-LIB [1] has no syntax for indicating the goal (unlike TPTP [20]). To address this we define a notion of *goal symbol* that is independent of the notion of an explicit goal being given.

Given a set of input formulas $F_1, \ldots, F_n$ and a set $\mathcal{G}$ containing zero or more formulas F_i marked as goal formulas, a *goal symbol* is a symbol such that

- It appears in a formula $F \in \mathcal{G}$, or
- It is a Skolem constant introduced in the clausification of some $F \in \mathcal{G}$, or
- It appears in at most *limit* formulas, or
- It is a Skolem constant introduced by the Skolemisation of some formula F_i of the form $\exists x.F$

where *limit* is a parameter to the process. In the case where this is a single goal formula we would expect *limit* to be 1. However, the input may have been subject to some additional preprocessing meaning that the goal is represented by a few clauses in the input. The last point is because many goals will take the form of negated universal statements; this is also how formulas for induction are identified in [15].

Once all such goal symbols have been identified, the set $\mathcal{G}$ is extended to include all formulas containing a goal symbol. This is done as $\mathcal{G}$ typically plays another role in proof search as clauses derived from formulas in $\mathcal{G}$ may be prioritised in clause selection, providing some heuristic goal-directionality.

5 Case Splitting for Free

An important part of inductive proofs is typically the case splitting between the base case and the inductive step. In this section we describe a clause splitting approach (implemented in Vampire as AVATAR [13, 21]) that achieves this.

We briefly describe the ground part of the AVATAR framework for clause splitting as case splitting for induction only requires the ground part. The general idea is that given a set of clauses S and a ground clause $L_1 \vee L_2$ we can consider the two sub-problems $S \cup \{L_1\}$ and $S \cup \{L_2\}$ independently.

Let name be a function from ground literals to labels that is injective up to variable renaming and symmetry of equality. Let $C \leftarrow A$ be a *labelled clause* where A is a set of labels. We can lift an inference system on clauses to one on labelled clauses where all conclusions take the union of the labels in premises. The previous rules for induction can be extended such that the consequent clauses take the labels of the premise clause.

Figure 2 shows how the simple saturation algorithm from Section 2 can be extended to perform ground clause splitting. It assumes a SAT procedure that we add clauses of labels to and then request the *difference* between a new model and the previous model in terms of added and removed labels.

```
input: Init: set of clauses;
var active, passive, unprocessed: set of clauses; var given, new: clause;
active := ∅;  passive := ∅;  unprocessed := Init;
loop
  while unprocessed ≠ ∅
    new:=pop(unprocessed);
    if new = □ then return unsatisfiable;
    if new = □ ← A then add ¬A to SAT;
    if new is ground then add label(new) to SAT;
    else add new to passive;
  if passive = ∅ then return satisfiable or unknown
  (add_labels, remove_labels) = new_model(SAT);          (* compute new model *)
  active:= {C ← L ∈ active | L ∩ remove_labels = ∅};
  passive:= {C ← L ∈ passive | L ∩ remove_labels = ∅};
  passive:=passive ∪ {retrieve(l) ← l | l ∈ add_labels};
  given := select(passive);                               (* clause selection *)
  move given from passive to active;
  unprocessed:=infer(given, active);                      (* generating inferences *)
```

Fig. 2. Simple Saturation Algorithm with Ground Clause Splitting.

To understand why this is very useful consider the conclusions of the inference rules given in Table 1. These clauses are all ground and multi-literal i.e. they capture multiple *cases*. As an example, when proving the conjecture $\mathsf{height}(t) \geq 0$ (for a suitable axiomatisation of height) our implementation considers and refutes between 6 and 8 different different cases depending on which form of induction rule is used.

6 Experimental Evaluation

In this section we describe the implementation and evaluation of the techniques described in this paper.

Implementation. We extended the Vampire [9] theorem prover with additional options to capture the techniques described in the previous sections. Table 2 gives an overview of these new options. The `sik` option captures the different approaches introduced in Section 3. The `indm` option limits the *depth* of induction. The remaining options capture the choices made in Section 4. Our implementation of the induction inference rules ensures that we never instantiate the same induction axiom more than once and that proof search when there are no term algebra sorts in the problem is unaffected. Furthermore, this implementation is fully compatible with all other proof search options and heuristics in Vampire. Our implementation is available online[3].

Experimental Setup. We use two sets of benchmarks from SMT-LIB from the UFDT and UFDTLIA logics where UF stands for Uninterpreted Functions, DT stands for DataTypes and LIA stands for Linear Integer Arithmetic; we do

[3] See `https://github.com/vprover/vampire`. Currently the functionality is in the branch `infp` but will soon be merged into `master`.

Table 2. New options and their values.

Name	Values	Description
`ind`	`none`, `struct`	Whether structural induction should be applied or not.
`sik`	1, 2, 3, `all`	The kind of structural induction to apply. The numbers 1,2, 3 refer to the three kinds introduced in Section 3 and `all` applies them all.
`indmd`	$n \geq 0$ (0)	The maximum *depth* to which induction is applied where 0 indicates it is unlimited.
`indc`	`goal`, `goal_plus`, `all`	Choices for the sel_α predicate (see Section 4) where `goal` uses goal symbols only, `goal_plus` uses goal and induction symbols, and `all` is unrestricted.
`indu`	`on`, `off`	Whether to include the sel_U predicate.
`indn`	`on`, `off`	Whether to include the sel_N predicate.
`gtg`	`on`, `off`	Whether goal clauses in the input should be inferred.
`gtgl`	$n \geq 1$ (1)	The *limit* of times a symbol should appear in input formulae to be identified as a goal symbol.

not consider AUFDTLIA as it does not contain problems interesting for induction. UFDT consists of 4376 problems *known not to be satisfiable* (we excluded problems either marked as, or found to be, satisfiable during experiments) and UFDTLIA consists of 303 problems. Experiments are run on StarExec [19].

6.1 Research Questions In this section we look at two research questions that naturally arise in our work.

Which options are useful? Given the set of introduced options, we would like to know which will be useful in general. Vampire is a portfolio solver and would normally run a series of strategies combining different options. Therefore, any options able to solve problems uniquely may be useful for a portfolio mode. Table 3 compares the option values across the SMT-LIB problems. All option values with the exception of `--sik three` and non-zero values for `indmd` solve some problems uniquely. For each option there is a clear choice for default value. The fact that non-zero values for `indmd` were not useful in general suggests that there was not a problem with an explosion of iterative induction steps. This is most likely due to the fact that clause selection will favour a breadth-first exploration of the space. The solved problems did not rely heavily on inferring goal symbols or selection via special symbols. This suggests that the problems of interest either had shallow proofs that followed quickly from the input, or contained few relevant symbols for induction.

What do the proofs look like? We ran Vampire in a portfolio mode using the additional options `-sik one -indm 0 -indc all` on the SMT-LIB UFDTLIA problem set and recorded (i) the number of induction steps appearing in proofs, and (ii) the maximum depth of these inductions. The results are in Table 4. In the majority of cases only a few induction steps are used but there are 11 problems where more than 10 inductions are required and the proof of `induction-vmcai2015/leon/heap-goal3.smt2` uses 145 induction steps. As suggested above, induction is relatively shallow with the maximum depth in proofs being 6 and most necessary inductions not being nested.

Table 3. Comparing option values.

Value	Count	Unique	Value	Count	Unique	Value	Count	Unique
`sik`			`indmd`			`indc`		
`one`	3088	20	0	3096	37	`all`	3069	104
`two`	3028	3	1	3044	0	`goal`	2989	7
`three`	3019	0	2	3051	0	`goal_plus`	2985	1
`all`	3043	2	3	3048	0			
`indu`			`indn`			`gtg`		
`on`	3095	43	`on`	3088	50	`on`	2992	27
`off`	3053	1	`off`	3046	8	`off`	3069	104

Table 4. Statistics from 165 successful problems in UFDTLIA.

Number of inductions in proof	Count
0	44
1	82
2	16
3	6
5	2
10-50	7
50-145	4

Max induction depth	Count
1	84
2	25
3	4
4	3
6	1

6.2 Comparative Evaluation We compare the new techniques to CVC4 on the SMT-LIB benchmarks in Table 5 running both solvers with and without induction. We currently restrict our attention to CVC4 as this is the only solver available that runs on these problems and supports induction (Z3 does not support induction). It is worth noting that CVC4 was reported comparable to Zipperposition in [4] but has improved considerably in the meantime.

Overall CVC4 solves more problems but Vampire solves 48 problems that CVC4 (or any other solver) does not. We consider it an impressive result for a first implementation and believe that Vampire will solve many more previously unsolved problems when more heuristics, options and induction axioms are implemented.

It is interesting to note that the majority of problems are solvable without induction, suggesting the need for better benchmarks. However, we also observe that Vampire will commonly use induction to solve a problem more quickly even when induction is not required. This is also a very interesting observation since normally the addition of new rules other than simplification slows down saturation theorem provers.

Table 5. Comparative results with CVC4 on SMT-LIB benchmarks.

Logic	Size	Solvers			
		CVC4-noi	Vampire-noi	CVC4	Vampire
UFDT	4376	2270	2226 (2)	2275 (5)	2294 (37)
UFDTLIA	303	69	76	224 (69)	165 (9)

7 Related Work

We focus on *explicit* induction approaches, rather than *implicit* induction, e.g. the inductionless induction [3] approach. Within this we identify two areas of relevant work - the specialised area of *inductive theorem proving* and the general approach of extending first-order theorem provers with induction.

Tools that use theorem provers as backends often include induction hypotheses in the input. For example, Dafny was extended to wrap SMT solvers with an induction layer inserting useful induction hypotheses [10] within Dafny.

There are a number of *inductive theorem provers* such as ACL2 [7], IsaPlanner [5], Zeno [18], and Hipspec [2]. These have special procedures for deciding when to apply induction and the main effort is in choosing appropriate points as the effort required for each application of induction is large. In general, such solvers are well suited to problems that require complex induction but only require relatively simple reasoning otherwise. Our focus is the converse case.

There main previous attempt to extend a saturation-based superposition theorem prover with induction is in Zipperposition by Cruanes [4]. This approach is formulated for (generally defined) structural induction over inductive datatypes. The main difference between this previous work and ours is the way in which this previous work puts together datatype reasoning, inductive reasoning, and reasoning by cases using AVATAR, whereas our work keeps all three parts separate. As a result, our approach is more general; our definition of induction does not depend on inductive datatypes and works without AVATAR, so it can be with little effort added to existing saturation theorem provers. For example, our generality results in the ability to implement well-founded induction.

Although we do note that Cruanes explores heuristics for *where to apply induction* from the broader inductive theorem proving literature that we have not yet explored.

Finally, we note that the experimental results of [4] have a different focus from our own as they focus on problems suited for inductive theorem provers whereas our research (and our experiments) focus on problems requiring a little bit of reduction and a lot of complex first-order reasoning.

Another approach [22] wraps superposition-based proof search in an extra process that iteratively explores the space of possible inductions. CVC4 has been extended with a set of techniques for induction [15]. There rules are similar to ours but the setting is different as CVC4 is a DPLL(T)-based SMT solver using quantifier instantiation to handle quantifiers.

8 Conclusion

In this paper we introduce a new method for integrating induction into a saturation-based theorem prover using superposition. Our approach utilises the clause-splitting framework for case splitting. Experimental results show that the new options allow us to solve many problems requiring complex (e.g. nested) inductions.

Acknowledgements. We thank Andrew Reynolds for helping with obtaining CVC4 results.

References

1. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
2. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 392–406, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
3. Hubert Comon. Inductionless induction. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 913–962. 2001.
4. Simon Cruanes. Superposition with structural induction. In *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings*, pages 172–188, 2017.
5. Lucas Dixon and Jacques Fleuriot. Higher order rippling in isaplanner. In *International Conference on Theorem Proving in Higher Order Logics*, pages 83–98. Springer, 2004.
6. Ashutosh Gupta, Laura Kovacs, Bernhard Kragl, and Andrei Voronkov. Extensional crisis and proving identity. In Franck Cassez and Jean-François Raskin, editors, *12th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8837 of *Lecture Notes in Computer Science*, pages 185–200. Springer, 2014.
7. Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
8. Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. *SIGPLAN Not.*, 52(1):260–270, January 2017.
9. Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of CAV*, volume 8044 of *LNCS*, pages 1–35, 2013.
10. K. Rustan M. Leino. Automating induction with an smt solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 315–331, Berlin, Heidelberg, 2012. Springer-Verlag.
11. Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.
12. Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 335–367. 2001.
13. Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 39–52. EasyChair, 2016.
14. Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016.
15. Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 80–98, 2015.
16. Tatiana Rybina and Andrei Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*, 2(2):155–181, 2001.
17. Stephan Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2-3):111–126, 2002.

18. William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–421. Springer, 2012.
19. Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec, a cross community logic solving service. `https://www.starexec.org`, 2012.
20. Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
21. Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer International Publishing, 2014.
22. Daniel Wand. *Superposition: Types and Induction. (Superposition : types et induction).* PhD thesis, Saarland University, Saarbrücken, Germany, 2017.
23. C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.