# Recursive Function Definitions in Superposition Reasoning

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

[1] TU Wien, Austria
[2] University of Manchester, UK
[3] EasyChair

**Abstract.** Program verification and analysis are two important areas where first-order quantified formulas and theories are used. Saturation-based theorem proving in this context provides an effective toolset for full automation but still lacks some of the more advanced techniques applied in state-of-the-art proof-assistants to tackle difficult inductive problems. We look at a slice of this domain characterized by datatypes and recursive function definitions and focus on (1) stronger induction formula generation based on function definitions and (2) inference rules suited for inductive problems that help during saturation. We implement these in the state-of-the-art superposition prover Vampire.

## 1 Introduction

This paper aims to contribute to the automation of proving problems with inductive types in the area of saturation-based theorem proving. Specifically, it introduces (1) new techniques to generate induction formulas and (2) new inference rules in saturation-based proof search to tackle some of the issues inherent in this topic.

Automation in inductive proofs started in the 70s with a sequence of theorem provers like ACL2 [?], CLAM/OYSTER [?] or IsaPlanner [?] just to name a few. In the nowadays more and more prominent field of saturation-based theorem proving, induction has been relatively out of focus with some work done in e.g. ZipperPosition [?], Vampire [?,?] or CVC4 [?,?]. These initial inductive techniques mostly utilize basic induction formulas for term algebra datatypes/integers and very little help is provided to the prover during saturation in order to actually solve the problems. We address several of these issues in this paper.

## 2 Motivating Examples

Finding an induction formula that leads to a successful proof for an inductive problem is known to be undecidable. Some provers like ACL2 or IsaPlanner take into account the structure of any function present in an inductive goal when creating induction formulas and provide heuristics to select the most suitable one

| | | | |
|---|---|---|---|
| $\forall y.0 + y$ | $:= y$ | $\forall x.\mathtt{nil} +\!\!+\, x$ | $:= x$ |
| $\forall x, y.\mathtt{s}(x) + y$ | $:= \mathtt{s}(x + y)$ | $\forall x, y, z.\mathtt{cons}(x, y) +\!\!+\, z$ | $:= \mathtt{cons}(x, y +\!\!+\, z)$ |
| $\mathtt{even}(0)$ | $:= \top$ | $\mathtt{fltn}(\mathtt{leaf})$ | $:= \mathtt{nil}$ |
| $\mathtt{even}(\mathtt{s}(0))$ | $:= \bot$ | $\forall u, v, w.\mathtt{fltn}(\mathtt{node}(u, v, w))$ | $:=$ |
| $\forall z.\mathtt{even}(\mathtt{s}(\mathtt{s}(z)))$ | $:= \mathtt{even}(z)$ | $\mathtt{fltn}(u) +\!\!+\, \mathtt{cons}(v, \mathtt{fltn}(w))$ | |
| $\forall x.0 \leq x$ | $:= \top$ | $\forall x.\mathtt{fltn}_2(\mathtt{leaf}, x)$ | $:= x$ |
| $\forall x.\mathtt{s}(x) \leq 0$ | $:= \bot$ | $\forall u, v, w, y.\mathtt{fltn}_2(\mathtt{node}(u, v, w), y)$ | $:=$ |
| $\forall x, y.\mathtt{s}(x) \leq \mathtt{s}(y)$ | $:= x \leq y$ | $\mathtt{fltn}_2(u, \mathtt{cons}(v, \mathtt{fltn}_2(w, y)))$ | |

**Fig. 1.** Recursive function definitions used

as the next proof step. Such heuristics are however still relatively uncommon in saturation-based theorem provers.

*Example 1.*

$$\forall x, y.(\mathtt{even}(x) \land \mathtt{even}(y)) \rightarrow \mathtt{even}(x + y)$$

The function definitions for `even` and + can be found in Figure 1. We can do the straightforward derivation:

| | |
|---|---|
| $\mathtt{even}(\sigma_0)$ | 1. input |
| $\mathtt{even}(\sigma_1)$ | 2. input |
| $\neg\mathtt{even}(\sigma_0 + \sigma_1)$ | 3. input |
| $\left(\begin{array}{c} \mathtt{even}(0 + \sigma_1)\land \\ \forall z.(\mathtt{even}(z + \sigma_1) \rightarrow \mathtt{even}(\mathtt{s}(z) + \sigma_1)) \end{array}\right) \rightarrow \forall x.\mathtt{even}(x + \sigma_1)$ | 4. induction f. |
| $\neg\mathtt{even}(0 + \sigma_1) \lor \mathtt{even}(\sigma_2 + \sigma_1)$ | 5. bin.res. 3, cnf(4) |
| $\neg\mathtt{even}(0 + \sigma_1) \lor \neg\mathtt{even}(\mathtt{s}(\sigma_2) + \sigma_1)$ | 6. bin.res. 3, cnf(4) |
| $\neg\mathtt{even}(\sigma_1) \lor \mathtt{even}(\sigma_2 + \sigma_1)$ | 7. 5, + axiom |
| $\neg\mathtt{even}(\sigma_1) \lor \neg\mathtt{even}(\mathtt{s}(\sigma_2) + \sigma_1)$ | 8. 6, + axiom |
| $\mathtt{even}(\sigma_2 + \sigma_1)$ | 9. bin.res. 2, 7 |
| $\neg\mathtt{even}(\mathtt{s}(\sigma_2) + \sigma_1)$ | 10. bin.res. 2, 8 |
| $\neg\mathtt{even}(\mathtt{s}(\sigma_2 + \sigma_1))$ | 11. 10, + axiom |
| ... | |
| $\mathtt{even}(\mathtt{s}(\sigma_3 + \sigma_1))$ | 12. |
| $\neg\mathtt{even}(\sigma_3 + \sigma_1)$ | 13. |

After selecting (3) for induction and binary resolving it with the clausal form of the simplest induction formula (4) with $\sigma_0$ as induction term, we get (5) and (6). Simplifications give (9) and (11) but there is no `even` axiom to further simplify. One more induction on $\sigma_2$ and similar simplifications yield (12) and (13). Neither of the hypotheses (9) and (12) match the conclusions (11) and (13), either due to the different Skolem constants or the different term structure.

Using $\sigma_1$ as induction term does not help either – we cannot get rid of any constructor terms in the second argument position of $+$.

In this paper, we create "matching" induction formulas using function definitions and then – if different induction formulas are generated for a goal – combine these in a way that all function terms can be simplified in the case distinction while keeping the necessary hypotheses.

In a saturation-based theorem prover, even the "correct" induction formula is useless if the term ordering prohibits rewriting function terms into their definitions or using necessary induction hypothesis, ultimately leading to a stuck proof.

*Example 2.* Given a unit-clause:

$$\{\mathtt{fltn}(\sigma_0) +\!\!+ \sigma_1 \neq \mathtt{fltn}_2(\sigma_0, \sigma_1)\}$$

Proving this literal inductively with the simplest case distinction gives a step case $\mathtt{node}(u, v, w)$ with induction hypotheses $u$ and $v$. Due to the large terms on the right-hand side of function definitions $\mathtt{fltn}$ and $\mathtt{fltn}_2$ for case $\mathtt{node}(u, v, w)$, the induction step conclusion cannot be simplified:

$$\mathtt{fltn}(\mathtt{node}(\sigma_2, \sigma_3, \sigma_4)) +\!\!+ \sigma_1 \neq \mathtt{fltn}_2(\mathtt{node}(\sigma_2, \sigma_3, \sigma_4), \sigma_1)$$

So even though the right induction formula was found we could not solve one of its cases because of limitations in the calculus. We address this issue in a way that still preserves relative completeness and some of the properties of simplifications.

## 3 Preliminaries

We assume familiarity with *standard multi-sorted first-order logic with equality.* Functions are denoted with $f$, $g$, $h$, predicates with $p$, $q$, $r$ and variables with $x$, $y$, $z$, $u$, $v$, $w$, possibly with indices. We reserve the notation $\sigma$, $\sigma_0$, $\sigma_1$, etc. for Skolem constants. A term is *ground* if it contains no variables. The notation $\overline{x}$ and $\overline{t}$ means tuples of variables and terms, respectively.

We use the standard logical connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ and quantifiers $\forall$ and $\exists$. *Atoms* are built inductively from terms and predicate symbols. Atoms and their negations are called *literals.* For a literal $l$, we use the notation $\overline{l}$ to denote its opposite sign literal. *Formulas* are built from connectives and atoms.

Additionally, a disjunction of literals is a *clause.* We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. We call every term, literal, clause or formula an *expression.* We use the notation $s \trianglelefteq t$ to denote that $s$ is a *subterm* of $t$ and $s \triangleleft t$ if $s$ is a *proper subterm* of $t$.

We may use the words *sort* and *type* interchangeably. We distinguish special sorts called *inductive sorts*, function symbols for inductive sorts called *constructors* and *destructors.* We require that the signature contains at least one constant constructor symbol for every inductive type. Such a symbol is called a *base constructor*, while non-constant ones are called *recursive constructors*. We call the ground terms built from the constructor symbols of a sort its *term algebra*. Semantically, each $n$-ary constructor $c$ has $n$ corresponding destructors $d_1, ..., d_n$. For any constructor term $c(t_1, ..., t_n)$ with root symbol $c$, the following holds:

$$\forall 1 \leq i \leq n. d_i(c(t_1, ..., t_n)) = t_i$$

Moreover, we usually axiomatise every term algebra with the *injectivity*, *distinctness*, *exhaustiveness* and *acyclicity* axioms. The inductive types we use in this paper are:

$$\begin{aligned} \texttt{nat} &:= 0 && \mid \texttt{s}(\texttt{p}(\texttt{nat})) \\ \texttt{lst} &:= \texttt{nil} && \mid \texttt{cons}(\texttt{head}(\texttt{nat}), \texttt{tail}(\texttt{lst})) \\ \texttt{btree} &:= \texttt{leaf} && \mid \texttt{node}(\texttt{left}(\texttt{btree}), \texttt{val}(\texttt{nat}), \texttt{right}(\texttt{btree})) \end{aligned}$$

An *interpreted symbol* is a function or predicate whose meaning is defined through axioms, e.g. $=$ is an interpreted symbol in first-order logic with equality. All other symbols are called *uninterpreted*. We distinguish *function/predicate definitions* from regular axioms. These define a branch of computation for a function/predicate. Such axiom is denoted by marking exactly one equality literal in it with $:=$ such as $F \rightarrow l := r$ which means that the orientation of this equality is fixed as left-to-right, $l$ is a function header and $r$ is a function definition, $F$ is the guard condition for this branch. We abuse this notation for predicate definitions where $:=$ can be replaced with a $\leftrightarrow$.

A notation we use is $E[s]$ meaning there is zero *TODO: is this okay with zero? used in IndM rule* (or more) distinguished occurrence(s) of the term $s$ in $E$. $E[t]$ then means that these occurrences are changed to a term $t$. We may abbreviate $E[t_1]...[t_n]$ with $E[\overline{t}]$.

## 3.1 Saturation-based proof search

Given a set of input formulas $C$ in clausal form, the set of all derivable clauses using an inference system from the set $C$ is called the *closure of* $F$ w.r.t. the system. If the closure contains $\square$, the original set $C$ is unsatisfiable, otherwise it is satisfiable. The process of computing the closure is called *saturation*. In practice more subtle notions are needed to tackle this problem, the first one is *saturation up to redundancy*. A clause $C$ is *redundant* w.r.t a set of clauses $S$ if some subset of $S$ of clauses smaller than $C$ w.r.t $\succ$ logically imply $C$. An inference system is usually equipped with *simplification and deletion rules* to get rid of redundant clauses. Second, selection methods are used which control the order in which the inferences are applied. For a more detailed discussion on saturation algorithms see [**?**].

We use the *superposition calculus* as the inference system in this paper. It works on sets of clauses – we denote the conversion of a formula $F$ to its *clausal normal form* with $\texttt{cnf}(F)$. The superposition calculus is *sound* and *refutationally* *complete*. A *refutation* is a derivation of $\bot$. Refutational completeness means for any unsatisfiable formula set, we can derive the empty clause. Therefore, with superposition we usually negate our input conjecture and try to refute it which, if successful, means the original conjecture is valid.

For the completeness a simplification term ordering $\succ$ of terms is needed (e.g. KBO, LPO) which is extended to literals and clauses with the multiset-extension. We will abuse notation and use the same symbol $\succ$ to denote the original ordering and its extensions.

We introduce an inference rule for induction adapted from [**?**]:

$$\frac{\neg L[t] \vee C}{\texttt{cnf}(F \to \forall y.(L[y]))}\ (\texttt{Ind})$$

where $L$ is a ground literal, $C$ is a clause and $F \to \forall y.(L[y])$ is some valid induction formula. The conclusion of the rule can be then resolved against its premise, yielding the clauses $\texttt{cnf}(\neg F) \vee C$.

## 4 Function definitions and Structural Induction

Interpreted functions are often axiomatized in a definitional form, meaning that given these axioms for a well-founded function $f$, any occurrence of $f$ in a ground function term can be completely eliminated. In this paper, we consider well-founded (terminating) functions that are *not mutually-recursive*. Although handling *recursive* functions needs a bit of precaution because expanding any non-ground recursive function term into its definition may lead to a rewrite loop, we treat non-recursive functions the same since they can be thought of as functions that contain only base cases.

### 4.1 Generating induction formulas from function definitions

A function definition is a set of axioms of the form

$$F \to f(s_1, ..., s_n) := t$$

as presented in Section 3. For now, we treat the detection of such function definitions as a black box and address this issue later. An argument position $1 \leq i \leq n$ of $f$ is an *active argument position* if there is an axiom like above of $f$ s.t. for some $f(t_1, ..., t_n) \trianglelefteq t$ and $s_i \neq t_i$. Given a literal $L$, we mark its *active subterms* inductively as follows:

- if $L$ is an equality $l = r$, then $l$ and $r$ are active subterms
- for an active recursive predicate/function subterm $f(s_1, ..., s_n)$ of $L$ with active argument positions $I$, all $s_i$ with $i \in I$ are active subterms of $L$
- for an active non-recursive predicate/function subterm $f(s_1, ..., s_n)$ of $L$, all $s_i$, $1 \leq i \leq n$ are active subterms of $L$
- if $c(t_1, ..., t_m)$ is an active constructor subterm of $L$ of type $\tau$, all $t_j$ with type $\tau$ are active subterms of $L$

*TODO: should we exclude non-recursive function terms here? it is sometimes good to induct on them (which is just a case distinction) but otherwise it can be confusing to introduce this here*

Every active subterm of a literal is a candidate for induction since replacing these terms with a case distinction of constructor terms leads to simplifiable literals that we can possible solve. Functions can recurse on multiple arguments, so instead of creating an induction formula for each active subterm, we look at active function subterms with non-empty active argument position sets and create induction formulas from their active subterms.

In order to create an induction formula, we create (1) a case distinction on the induction terms that is exhaustive and non-overlapping (and can contain guard conditions) and (2) induction hypotheses for the step cases. For both, we substitute the induction terms with the terms from the corresponding case/hypothesis into the original literal. For the case distinction, we use the function axioms which are by assumption well-defined.

Sometimes function terms contain only complex terms in their active argument positions and generalizing over them does not always lead to a proof. E.g. in $s \leq t+s$, both arguments of the outermost function $\leq$ are active but only the first argument of $+$ is active, therefore it is essential to induct on both $s$ and $t$. The other possibility is to induct on $t+s$ from the second position which would render the original true literal false.

To solve this issue, we create tuples of possible induction terms for each such function term by taking every combination of active subterms of $s_i$ for all $i$ in the set of active argument positions. In the above example, this yields $(s,t)$ and $(s, t+s)$ as possible induction term tuples. We can create from them and the axioms of $\leq$ the following case distinctions by substituting each tuple element with the corresponding argument in the function's axioms in the literal $s \leq t+s$:

$$\begin{array}{lll}
\forall x.0 \leq x := \top & \Rightarrow \quad 0 \leq x + 0 & (L_1)\\
\forall x.\mathsf{s}(x) \leq 0 := \bot & \Rightarrow \quad \mathsf{s}(x) \leq 0 + \mathsf{s}(x) & (L_2)\\
\forall x,y.\mathsf{s}(x) \leq \mathsf{s}(y) := x \leq y & \Rightarrow \quad \mathsf{s}(x) \leq \mathsf{s}(y) + \mathsf{s}(x) & (L_3)
\end{array}$$

The next step is to add induction hypotheses where possible – an axiom $F \rightarrow f(s_1, ..., s_n) := t$ gives induction hypotheses for the corresponding case in the case distinction if there are recursive calls in $t$. Each such recursive call gives rise to a new literal the same way as in the case distinctions. For $\leq$, only the last case has a recursive call:

$$\forall x,y.\mathsf{s}(x) \leq \mathsf{s}(y) := x \leq y \qquad \Rightarrow \qquad x \leq y + x \qquad (L_{31})$$

The obtained literals are then conjuncted to get the induction formula antecedent. The induction hypotheses for each case are conjuncted then added as antecedent to the literal of that case. We add a conclusion literal to the final formula where each induction term is replaced with a fresh variable:

$$\big(\forall x.L_1 \wedge \forall x.L_2 \wedge \forall x,y.(L_{31} \rightarrow L_3)\big) \rightarrow \forall u,v.u \leq v + u$$

The substitutions we apply to the literal may not be confluent or well-defined. For example, a literal $r \leq r + r$ would require us to substitute both 0 and $\mathtt{s}(x)$ for $r$ in the second case of $\leq$. The two terms cannot be unified so they would result in a literal we cannot solve and we discard this case. Hypotheses can also be discarded because the substitutions are inconsistent. Losing hypotheses this way is not an issue, but losing a case from the induction formula can result in unsoundness. *TODO: state that case distinctions are well-defined even when we throw away cases?*

## 4.2 Containment and union of induction formulas

Even though there may be multiple induction formulas used for the proof of an inductive goal, the process described in Section 4.1 can create many similar or redundant ones. Therefore an important step is to discard the unusable ones and combine the rest if possible. Looking at the Example 1 once again, from the literal $\neg\mathtt{even}(\sigma_0 + \sigma_1)$ we can create two induction formulas with non-complex induction terms. One is generated from $\mathtt{even}(\sigma_0 + \sigma_1)$:

$$\begin{pmatrix} \mathtt{even}(0 + \sigma_1) \wedge \mathtt{even}(\mathtt{s}(0) + \sigma_1) \wedge \\ \forall z.(\mathtt{even}(z + \sigma_1) \rightarrow \mathtt{even}(\mathtt{s}(\mathtt{s}(z)) + \sigma_1)) \end{pmatrix} \rightarrow \forall u.\mathtt{even}(u + \sigma_1) \tag{1}$$

The other is given by $\sigma_0 + \sigma_1$:

$$\big(\mathtt{even}(0 + \sigma_1) \wedge \forall x.(\mathtt{even}(x + \sigma_1) \rightarrow \mathtt{even}(\mathtt{s}(x) + \sigma_1))\big) \rightarrow \forall z.\mathtt{even}(z+\sigma_1) \tag{2}$$

Practically, trying (2) is a waste of resources as we have already seen in Section 2 so we better off discarding it. There are very simple approaches in the literature on which one can base this decision. E.g. ACL2 would discard it because the step case of (1) is in the *transitive closure* of that of (2). In terms of well-founded relations, (2) is based on the relation $x \prec \mathtt{s}(x)$ and applying this twice gives us $x \prec \cdot \prec \mathtt{s}(\mathtt{s}(x))$, that is, the relation for (1).

The reason this works is only incidental – although the case distinction of $+$ is more general than that of $\mathtt{even}$ and therefore every term in the case distinction of $\mathtt{even}$ will match a function definition of $+$, the induction hypothesis of the two are completely unrelated and therefore discarding one of them is like flipping a coin before the proof without knowing which one will be needed.

In order not to lose useful structure from the generated induction formulas, our goals are the following: when discarding or combining induction formulas (1) more special case distinctions must be preserved so that all considered function definitions are matched and (2) all induction hypotheses must be preserved since a priori we do not know which ones will be used.

For two induction formulas $F_1$ and $F_2$ with identical case distinctions, $F_1$ *contains* $F_2$ if for each case, the set of induction hypotheses of the case in $F_1$ is a superset of that of $F_2$. If an induction formula is contained by another, we can safely discard it since its case distinction and induction hypotheses can be found in the second.

*Example 3.* *TODO: add example*

If two induction formulas do not contain each other, we can either use them separately or use their *union* instead [**?**]. The union of two induction formulas provides a case distinction with all cases from the original ones while supplying the necessary induction hypotheses. We can compute the union with the following two approaches:

1. We can think of the two in terms of well-founded orders and for each pair of subrelations from the first and the second, we compute their intersections. Then, we check whether any subrelation has cases left "outside" the intersections. Finally, we convert this union of orders back to a formula by adding the remaining cases, i.e. the base cases.
2. Otherwise, we can simply intersect every case of the two induction formulas putting together their induction hypotheses.

*Example 4.* Let us look at the case distinction of formulas (1) and (2) from our running Example 1.

*Approach 1.* The formulas give relations $z \prec \mathtt{s}(\mathtt{s}(z))$ and $x \prec \mathtt{s}(x)$, respectively. The intersection of the two is a relation $z, \mathtt{s}(z) \prec \mathtt{s}(\mathtt{s}(z))$. This can be seen as letting the more general $\mathtt{s}(x)$ to match only its $\mathtt{s}(\mathtt{s}(z))$ instance and putting together the related terms $z$ and $x$, but since $\mathtt{s}(x)$ was specialized to $\mathtt{s}(\mathtt{s}(z))$ we also need to specialize $x$ $\mathtt{s}(z)$.

Then, we need to check what remains of the original subrelations apart from the intersection. While $z \prec \mathtt{s}(\mathtt{s}(z))$ was used as-is in the intersection, so it is covered, an instance of $x \prec \mathtt{s}(x)$ was used, thus the remaining instance $0 \prec \mathtt{s}(0)$ is added.

Converting this back to an induction formula, to make sure the case distinction is complete, we add base case 0 which is not covered by $\mathtt{s}(0)$ nor $\mathtt{s}(\mathtt{s}(z))$.

*Approach 2.* We take intersections of each case from (1) and (2) pairwise:

- 0 and 0 overlap in 0. Both are base cases so this results in a base case.
- 0 and $\mathtt{s}(x)$ do not overlap.
- $\mathtt{s}(0)$ and 0 do not overlap.
- $\mathtt{s}(0)$ and $\mathtt{s}(x)$ overlap in $\mathtt{s}(0)$. The second is a recursive case, so we add the induction hypothesis 0 which is $x$ specialized to match the overlap.
- $\mathtt{s}(\mathtt{s}(z))$ and 0 do not overlap.
- $\mathtt{s}(\mathtt{s}(z))$ and $\mathtt{s}(x)$ overlap. Both are recursive cases, so we add the two induction hypotheses, specialized where needed ($x$ is specialized to $\mathtt{s}(z)$).

Both approaches give the induction formula:

$$\begin{pmatrix} \mathtt{even}(0+\sigma_1) \wedge (\mathtt{even}(0+\sigma_1) \rightarrow \mathtt{even}(\mathtt{s}(0)+\sigma_1)) \wedge \\ \forall z.((\mathtt{even}(z+\sigma_1) \wedge \mathtt{even}(\mathtt{s}(z)+\sigma_1)) \rightarrow \mathtt{even}(\mathtt{s}(\mathtt{s}(z))+\sigma_1)) \end{pmatrix} \rightarrow \forall u.\mathtt{even}(u+\sigma_1) \tag{3}$$

We note here that it is enough to check pairwise intersections only because the induction formulas (and the functions they are generated from) are by assumption well-defined, so each case distinction is non-overlapping and exhaustive.

One of the advantages of taking the union is that by definition it is guaranteed that each case matches exactly one function axiom for each of the original function terms and the necessary induction hypotheses are also always present. The disadvantages are that it may not be well-founded and that it can create very large induction formulas. It is also not clear when to use it. There are situations when two induction formulas use completely different induction terms. For example, the commutativity of +

$$\forall x, y. x + y = y + x$$

can be solved by first inducting on $x$ and then on $y$ or vice versa – the order is irrelevant. However, in general, depending on the order of simplifications and the axioms used for each function, inducting sequentially on multiple terms can lead to different results.

*TODO: Give some heuristics on when to use the union.*

## 4.3 Well-foundedness and well-definedness

So far we assumed that every function we are provided with is well-founded and well-defined. Functions that fail to meet this requirement are potential sources of unsoundness. Well-foundedness must be also checked when creating the union of two induction formulas.

We chose easy-to-check sufficient conditions for both. We base our well-foundedness check on a lexicographic order on the active argument positions of each function and use the subterm relation as elementary property. This can be easily checked with e.g. constructor-style input functions where arguments of inductive sorts on the left-hand side of a function definition are either variables or term algebra constructors. A common example for this is the Ackermann function given by:

$$\begin{aligned}
\forall y. \mathtt{ack}(0, y) &:= \mathtt{s}(y) \\
\forall x. \mathtt{ack}(\mathtt{s}(x), 0) &:= \mathtt{ack}(x, \mathtt{s}(0)) \\
\forall x, y. \mathtt{ack}(\mathtt{s}(x), \mathtt{s}(y)) &:= \mathtt{ack}(x, \mathtt{ack}(\mathtt{s}(x), y))
\end{aligned}$$

Any function term stemming from a function that is not well-founded is ignored while generating induction formulas.

After generating the union of two induction formulas, we check whether each case - recursive call pair obeys a lexicographic order and discard the union if not.

For well-definedness, we distinguish *under-definedness* and *over-definedness*. The former can lead to missing false cases, thus introducing unsoundness but it can be easily mitigated by adding those missing cases. The latter is a bit more complicated: an induction formula with overlapping cases is not necessarily unsound but it can lead to ill-constructed case distinctions for the union. *TODO: anything to add here?*

## 4.4 Induction hypothesis strengthening

Sometimes an inductive step fails because the required hypothesis only partially matches the conclusion after simplification. This can be resolved in some cases by adding more related pairs to the subrelation in the corresponding well-founded order.

As discussed previously, a suitable lexicographic order establishes the well-foundedness of any function. This order determines which induction terms are needed for the current subrelation to be decreasing. Any other induction terms in the relation can be strengthened to relate all possible pairs by replacing their terms in the hypothesis with fresh variables. Other than that, non-induction terms – which can be thought of as induction terms implicitly stated and not needed for the well-foundedness – can be also strengthened this way.

*Example 5.* An induction formula on literal $L[s][t][r]$ based on the function `ack` with induction terms $s$ and $t$ and non-induction term $r$ is the following:

$$\begin{pmatrix} \forall y.L[0][y][r] \land \forall x.(L[x][\mathtt{s}(0)][r] \to L[\mathtt{s}(x)][0][r]) \\ \forall x,y.((L[x][\mathtt{ack}(\mathtt{s}(x),y)][r] \land L[\mathtt{s}(x)][y][r]) \to L[\mathtt{s}(x)][\mathtt{s}(y)][r]) \end{pmatrix} \to \forall u,v.L[u][v][r]$$

We marked the occurrences in each hypothesis with red that are not needed for the well-foundedness. These can be replaced with universally quantified fresh variables:

$$\begin{pmatrix} \forall y.L[0][y][r] \land \forall x.(\forall z_0,z_1.L[x][z_0][z_1] \to L[\mathtt{s}(x)][0][r]) \\ \forall x,y.((\forall z_2,z_3.L[x][z_2][z_3] \land \forall z_4.L[\mathtt{s}(x)][y][z_4]) \to L[\mathtt{s}(x)][\mathtt{s}(y)][r]) \end{pmatrix} \to \forall u,v.L[u][v][r]$$

## 4.5 Generalizing over occurrences and complex terms

*TODO: maybe shorten this to referencing the generalization paper and add that we always include all active term occurrences* A useful general proof technique is *generalizing formulas*, i.e. proving a more general statement. This is usually done to obtain shorter proofs or get rid of some complexity which prevents us from getting a proof in the first place.

*Example 6.* The following statement cannot be solved by simply inducting on $x$:

$$\forall x.x \leq x + x$$

After simplifying the inductive step case

$$\forall x.x \leq x + x \to \mathtt{s}(x) \leq \mathtt{s}(x) + \mathtt{s}(x)$$

the conclusion and the hypothesis do not match:

$$\forall x.x \leq x + x \to x \leq x + \mathtt{s}(x)$$

We can generalize the original statement to leave out the problematic third occurrence of $x$ and induct on $z$ instead:

$$\forall x,z.z \leq z + x$$

Another example is the following, where simply inducting on $z$ leads to a non-theorem in the inductive step:

$$\forall y, z.\texttt{rev}(\texttt{rev}(z) +\!\!+ \texttt{cons}(y, \texttt{nil})) = \texttt{cons}(y, \texttt{rev}(\texttt{rev}(z)))$$

The solution is to generalize over $\texttt{rev}(z)$ to get a statement that is easier to prove:

$$\forall x, y.\texttt{rev}(x +\!\!+ \texttt{cons}(y, \texttt{nil})) = \texttt{cons}(y, \texttt{rev}(x))$$

It is usually beneficial to generalize any formula before inducting on it. However, generalizing over all possible occurrences of any subterm of a formula gives a large number of induction formulas, most of which is not provable. Therefore certain heuristics are needed to reduce the number of generated induction formulas. One is based on the observation that non-active subterm occurrences – when replaced with some constructor term in e.g. an inductive step conclusion – remain fixed and cannot be eliminated.

However, sometimes they are needed to preserve validity of the property we are trying to prove. On the other hand, active subterm occurrences are usually needed to prove the property. So our heuristics are mostly good-to-go if they generalize over the active occurrences and try out all combinations of non-active ones.

For a complex subterm, our heuristic is to generalize over it if it has at least one active occurrence. The individual occurrences are then generalized according to the rules presented above.

## 5 Function definition rewriting in Superposition

As discussed in Section 2, any function definition $f(\overline{x}) := t$ may be oriented in a different way in a superposition theorem prover due to $t \succ f(\overline{x})$ given by the ordering. The choices we have here is either not being able to simplify induction formulas or losing completeness.

One solution is to choose an ordering that orients all function definitions according to their intended usage [**?**]. However, even if the initial orientation exists, there may be new function definitions derived from the initial ones that need to be oriented on the fly which is not practically achievable. Otherwise, conflicting orientations of two variants for the same function definition can cause an infinite chain of rewriting.

Another solution is to allow rewriting according to the intended function definition orientation and at the same time disallow any inferences that would rewrite the function definitions themselves, creating new variants. Despite losing completeness in this case, depending on factors like whether we use constructor- or destructor-style function definitions, it can actually help discard inferences which we would not be used anyways, e.g. any instantiated variant for a function definition just creates redundancy in the search space.

We use this latter solution and we disallow any inferences with such function definition clauses except for rewriting with two special inference rules: one is a modified *paramodulation*, the other is a special case of *demodulation*. We let the special demodulation happen only if the function definition is a unit clause and it has an orientation which is simplifying w.r.t. the ordering $\succ$:

$$\frac{f(\overline{x}) := t \quad \cancel{L[f(\overline{x})\theta] \vee D}}{L[t\theta] \vee D}\ (\texttt{DemF}) \tag{4}$$

where $f(\overline{x})\theta \succ t\theta$ and $L[f(\overline{x})\theta] \vee D \succ f(\overline{x})\theta = t\theta$. Since for functions containing no mutually recursive functions, the function "dependency tree" is a DAG, it is expected that this rule can be used at least for the base cases of any function definition since those definitions can be oriented in the right way by simply creating a precedence from the DAG.

Second, all other non-simplifying rewriting with function definitions is done with the modified paramodulation rule:

$$\frac{f(\overline{x}) := t \vee C \quad L[f(\overline{x})\theta] \vee D}{L[t\theta] \vee C\theta \vee D}\ (\texttt{ParF}) \tag{5}$$

This rule has no side conditions so that we can rewrite any part of a clause which helps expand function headers when needed but at the same time we lose properties such as fairness and completeness.

Note that we only use clauses for such rewriting that contain exactly one equality literal marked as a function definition. One might define e.g. non-determinism with clauses containing more than one equality:

$$\texttt{f}(\overline{x}) = t_1 \vee \texttt{f}(\overline{x}) = t_2$$

We avoid handling such clauses in this manner as it is unclear how the other literal should be used in the resulting clause when we paramodulate with one of them.

## 6 Induction in Saturation-based proof search

Let us turn our attention to how induction is applied in saturation. We already introduced an induction rule (3.1) in Section 3 which only works with a single premise. Now we introduce a new inference rule which gives more freedom in applying induction – with the cost of increasing the search space. To see the motivation for this rule, let us go back to Example 1.

After creating formula (3) to fit both functions, we face yet another issue – we cannot refute the second case $\texttt{s}(0)$, in particular, this clause:

$$\neg\texttt{even}(\texttt{s}(\sigma_1))$$

This is not because the induction formula is invalid but because the original literal $\texttt{even}(\sigma_0 + \sigma_1)$ is not refutable by itself. In terms of the original formula, this case is true because the antecedent is false:

$$\forall y.(\texttt{even}(\texttt{s}(0)) \wedge \texttt{even}(y)) \to \texttt{even}(\texttt{s}(0) + y)$$

In the saturation, there is no literal corresponding to $\texttt{even}(\texttt{s}(0))$ which would make it refutable. Our proposal is the following: when assumptions in different clauses are missing to prove a literal by induction, we induct on all clauses at once. This can be done by creating the same induction formula but this time in place of the single literal, we use $\texttt{even}(\sigma_0) \to \texttt{even}(\sigma_0 + \sigma_1)$ (denoted by $L[\sigma_0]$):

$$L[0] \wedge (L[0] \to L[\texttt{s}(0)]) \wedge \forall x.(L[x] \wedge L[\texttt{s}(x)] \to L[\texttt{s}(\texttt{s}(x))]) \to \forall u.L[u]$$

The clauses resulting from this induction formula are binary resolved with both $\texttt{even}(\sigma_0)$ and $\neg\texttt{even}(\sigma_0 + \sigma_1)$, eventually leading to the empty clause (proof omitted).

We now present the general inference rule:

$$\frac{L_1[\overline{t}] \vee C_1 \quad \ldots \quad L_n[\overline{t}] \vee C_n \quad \neg L[\overline{t}] \vee C}{\texttt{cnf}(F \to \forall \overline{y}.(\bigwedge_{1\leq i\leq n} L_i[\overline{y}] \to L[\overline{y}]))} \ (\texttt{IndM})$$

where $L$ and $L_i$, $1 \leq i \leq n$ are ground literals, $C$ and $C_i$, $1 \leq i \leq n$ are clauses and $F \to \forall \overline{y}.(\bigwedge_{1\leq i\leq n} L_i[\overline{y}] \to L[\overline{y}])$ is a valid induction formula. $\overline{y}$ is a tuple of variables and $\overline{t}$ is a tuple of induction terms, both of size $m$. We once again binary resolve each resulting clause with the original literals to get $\texttt{cnf}(\neg F) \vee \bigvee_{1\leq i\leq n} C_i \vee C$.

**Theorem 1.** *The inference rule* `IndM` *is sound.*

*Proof. TODO: I think in this formulation of the rule, this is trivial since we can add any valid induction formula. Is it worth to state?*

*TODO: Elaborate on how to use this rule in practice, maybe add the rev-rev proof as a possible application.*

## 7 Experiments

We implemented the discussed techniques in Vampire and experimented them on a large number of benchmarks. In particular, we used a generated set of 3000+ problems of increasing difficulty which we call the Vampire Induction Benchmarks (VIB for short). This explicitly contains the function definition orientations via the `define-fun` and `define-fun-rec` keywords of SMT-LIB thus eliminating the possibility of errors that may arise from ill-oriented function definitions. We also looked at the UFDTLIA set of SMT-COMP that contains some well-known hard problems of inductive reasoning which required function definition discovery. *TODO: Explain somewhere function definition discovery or leave it out completely.*

The experiments were conducted using BenchExec [?,?] and each solver instance was provided one processor core, 16 GB of memory and a time limit of 300 seconds. Due to the large number of introduced flags, instead of showing individual results for each discussed technique, we present the results obtained by creating a custom schedule in *portfolio mode* with all combinations of flags (a total of 64 combinations). The results are shown in Figure 2. Vampire refers to the original Vampire, Vampire$^{**}$ to the new Vampire with only the new induction variant enabled and Vampire$^{\dagger}$ to Vampire$^{**}$ with the portfolio mode schedule. We measured two solvers ZipperPosition and CVC4, which showed the best results among all solvers for the VIB and UFDTLIA set, respectively.

## 8 Related Work

## 9 Conclusions

## Acknowledgements

| Problems | | Count | Solved problems | | | | |
|---|---|---|---|---|---|---|---|
| | | | VAMPIRE | VAMPIRE** | VAMPIRE$^{\dagger}$ | CVC4 | ZIPPERPOSITION |
| `lst` | ++ | 313 | 72 | 238 | 313 | 1 | 312 |
| | `pref` | 696 | 94 | 687 | 696 | 1 | 583 |
| | `rev` | 4 | 2 | 2 | 3 | 0 | 2 |
| `nat` | + assoc | 312 | 81 | 261 | 312 | 1 | 312 |
| | + mix | 830 | 103 | 263 | 591 | 70 | 552 |
| | + comm | 2 | 2 | 2 | 2 | 1 | 2 |
| | +s0 mix | 486 | 81 | 124 | 281 | 87 | 186 |
| | $\cdot$ | 6 | 0 | 0 | 2 | 0 | 0 |
| | `even` | 2 | 0 | 0 | 1 | 0 | 0 |
| | $\simeq$ | 4 | 2 | 3 | 3 | 2 | 4 |
| | $\leq$ | 696 | 117 | 696 | 696 | 1 | 583 |
| | `dup` | 1 | 0 | 0 | 0 | 0 | 0 |
| `btree` | `fltn` | 3 | 0 | 0 | 2 | 1 | 2 |
| `nat&lst` | +& ++ | 1 | 1 | 1 | 1 | 0 | 0 |
| UFDTLIA | | 327 | 144 | 149 | 176 | 214 | - |
| All | | 3683 | 699 | 2426 | 3079 | 379 | 2538 |

| Problems | | Count | Uniquely solved problems | | | | |
|---|---|---|---|---|---|---|---|
| | | | VAMPIRE | VAMPIRE** | VAMPIRE$^{\dagger}$ | CVC4 | ZIPPERPOSITION |
| `lst` | ++ | 313 | 0 | 0 | 1 | 0 | 0 |
| | `pref` | 696 | 0 | 0 | 0 | 0 | 0 |
| | `rev` | 4 | 0 | 0 | 1 | 0 | 0 |
| `nat` | + assoc | 312 | 0 | 0 | 0 | 0 | 0 |
| | + mix | 830 | 2 | 0 | 116 | 0 | 93 |
| | + comm | 2 | 0 | 0 | 0 | 0 | 0 |
| | +s0 mix | 486 | 0 | 0 | 95 | 0 | 3 |
| | $\cdot$ | 6 | 0 | 0 | 2 | 0 | 0 |
| | `even` | 2 | 0 | 0 | 1 | 0 | 0 |
| | $\simeq$ | 4 | 0 | 0 | 0 | 0 | 0 |
| | $\leq$ | 696 | 0 | 0 | 0 | 0 | 0 |
| | `dup` | 1 | 0 | 0 | 0 | 0 | 0 |
| `btree` | `fltn` | 3 | 0 | 0 | 0 | 0 | 0 |
| `nat&lst` | +& ++ | 1 | 0 | 0 | 0 | 0 | 0 |
| UFDTLIA | | 327 | 2 | 0 | 9 | 50 | - |
| All | | 3683 | 4 | 0 | 225 | 50 | 96 |

**Fig. 2.** VAMPIRE with and without portfolio mode and other solvers compared