Induction with Recursive Definitions in Superposition

Marton Hajdu, Petra Hozzová, Laura Kovacs and Andrei Voronkov

September 1, 2021

Formal Methods in Computer-Aided Design 2021

Induction with Recursive Definitions in Superposition

Márton Hajdu*, Petra Hozzová*, Laura Kovács* and Andrei Voronkov†
*TU Wien
†University of Manchester and EasyChair

***Abstract*—Functional programs over inductively defined data types, such as lists, binary trees and naturals, can naturally be defined using recursive equations over recursive functions. In first-order logic, function definitions can be considered as universally quantified equalities. Verifying functional program properties therefore requires inductive reasoning with both theories and quantifiers. In this paper we propose new extensions and generalizations to automate induction with recursive functions in saturation-based first-order theorem proving, using the superposition calculus. Instead of using function definitions as first-order axioms, we introduced new simplification rules for treating function definitions as rewrite rules. We guide inductive reasoning and strengthen induction schema using recursively defined functions. Our experimental results show that handling recursive definitions in superposition reasoning significantly improves automated reasoning with induction.**

I. Introduction

Automated reasoning has become the backbone of formal software development [1]. Automating inductive reasoning is of increasing importance for emerging applications in software verification, in particular in the context of functional programming and inductive/algebraic data types (also called term algebras), such as natural numbers, lists and binary trees. Functional programs can be typically described by recursive equations/functions over algebraic data types, as illustrated in Figure 1. On the other hand, algebraic data types are, for example, commonly used in security applications to encode uniqueness of hash functions [2] or to express non-interference properties preventing information flow between private/public channels [3]. Formalizing such properties requires full first-order logic with theories, and automating their validation requires inductive reasoning.

Previous works on automating induction mainly focus on inductive theorem proving [4], [5], [6], [7], [8], [9], [10], [11]: deciding when induction should be applied and what induction axiom should be used. Further restrictions are made on the logical expressiveness, for example induction over only universal properties [7], [9], [6], term algebras [12] or Horn clauses [13]. Recent advances related to automating inductive reasoning, such as first-order reasoning with inductively defined data types [14], inductive strengthening of SMT properties [15], structural induction in first-order theorem proving [16], [17], [18], [12], open up new possibilities for automating induction. *In this paper we focus on first-order theorem proving and automate induction by integrating it directly into the proof search algorithm of first-order theorem proving.* The program assertions from lines 17–18 of Figure 1 show what we strive for: validating first-order properties over algebraic data types, such as binary trees, lists and naturals, involving additional recursive function definitions and predicates, such as `even`, `mul`, `app`, `flat` and `aflat`. We prove such and similar inductive properties by using saturation-based proof search based on the superposition calculus [19], which is the leading technology in automated theorem proving [20], [21], [22].

Reasoning about inductively defined data types with recursive definitions. Our work targets full and efficient automation of induction with recursive function reasoning, as illustrated in a toy ML-like functional program of Figure 1. Lines 1–3 of Figure 1 declare respectively the algebraic data types of natural numbers `nat`, lists `list` and binary trees `bt`, using constructors. In first-order logic, these data types correspond to term algebras [14]. Functional programs over data types can be defined by recursive equations, for example lines 4-5 of Figure 1 define the addition `add` of two natural numbers x, y (in first-order logic, function definitions can be considered as universally quantified equalities). Verifying the correctness of Figure 1 requires then to prove the formulas of lines 17-18, which asserts the equivalence of two functions over binary trees (line 17) and even properties of naturals (line 18). Automating reasoning about properties of inductively defined data types like `nat`, `list` and `bt` needs to handle acyclicity already for equational properties (which, in general, is not finitely axiomatizable) and induction. Our recent results on reasoning with inductively defined data types and induction [14], [18] enable induction in superposition-based theorem proving, yet only by applying induction over one clause at a time. Our work builds upon these results and brings novel extensions for handling recursive functions and (generalized) induction on arbitrarily many clauses simultaneously.

Our contributions. This paper brings the following contributions.

- We introduce an induction formula generation method, utilizing unification and recursive function definitions over algebraic data types (Section IV). We propose inductive strengthening and generalization methods well-suited for saturation-based approaches.
- We propose new inference rules for induction in superposition by treating recursive function definitions over algebraic data types as rewrite rules in superposition (Section V). Moreover, we make use of induction hypotheses with specialized inference rules. Applications of induc-

https://doi.org/

```
1  datatype nat = zero | s of nat
2  datatype list = nil | cons of nat list
3  datatype bt = leaf | node of bt nat bt

4  add zero y = y
5  add (s x) y = s (add x y)
6  mul zero y = zero
7  mul (s x) y = add (mul x y) y
8  even zero
9  ¬even (s zero)
10 even (s (s x)) ↔ even x

11 app nil z = z
12 app (cons x y) z = cons x (app y z)
13 flat leaf = nil
14 flat (node x y z) = app (flat x) (cons y (flat z))
15 aflat leaf u = u
16 aflat (node x y z) u = aflat x (cons y (aflat z u))

17 assert (∀x, y)(app (flat x) y = aflat x y)
18 assert (∀x, y)(even y → even (mul x y))
```

Fig. 1. Motivating example with recursive definitions over algebraic data types.

tion become inference rules of the saturation process, adding instances of appropriate induction schemata.

- We extend superposition-based equational reasoning with new inference rules capturing inductive steps over multiple clauses and optimize saturation-based proof search with induction (Section VI). Unlike [16], our results do not necessarily depend on the AVATAR clause splitting framework [23]. Contrarily to [12], we are not limited to induction over term algebras with the subterm ordering and we stay in a standard saturation framework.
- We implemented our approach in the VAMPIRE theorem prover [22] and evaluated it on a large collection of examples, including 327 examples from the SMT-LIB repository [24] and 3,397 mathematical properties over naturals, lists and binary trees (Section VII).
- Our experiments show the potential of our new approach, by solving 527 problems that other systems automating induction could not prove (Section VII).

Structure of the paper. The rest of the paper is organized as follows. We illustrate the challenges of automating induction with recursive definitions in superposition reasoning in Section II. We present our induction formula generation method in Section IV. Section V describes inductive reasoning with recursive definitions, whereas Section VI generalizes our work to induction with multiple premises. After summarizing our experimental findings in Section VII, we overview related work in Section VIII. We conclude the paper in Section IX.

II. MOTIVATING EXAMPLE

We first motivate our work using the functional program of Figure 1 over naturals, lists and binary trees.

Example 1 (Inductive reasoning with lists and binary trees)**.** Using the recursive function definition app over lists, and recursive function definitions flat and aflat over binary trees (lines 11–16 of Figure 1), we first focus on proving the equivalence of functions flat and aflat flattening binary trees to lists, specified as an assertion at line 17 of Figure 1. For easing readability, we write this assertion in infix notation as below:

$$\forall u, v.\mathtt{app}(\mathtt{flat}(u), v) = \mathtt{aflat}(u, v) \tag{1}$$

Proving (1) requires induction over binary trees, using for example the structural induction formula

$$\begin{aligned}\big(F[\mathtt{leaf}] \land \forall x, y, z.\big((F[x] \land F[z]) \\ \to F[\mathtt{node}(x, y, z)]\big)\big) \to \forall u.F[u],\end{aligned} \tag{2}$$

where $F[x]$ denotes a first-order formula over x. By instantiating (2), proving (1) reduces to proving two formulas: the base case and the step case. The base case,

$$\forall v.\mathtt{app}(\mathtt{flat}(\mathtt{leaf}), v) = \mathtt{aflat}(\mathtt{leaf}, v), \tag{3}$$

holds by the recursive definitions at lines 11, 13 and 15 of Figure 1. For the step case, we strengthen the hypotheses by replacing v with fresh universally quantified variables v_0, v_1:

$$\forall x, y, z, v.\big(\forall v_0.\mathtt{app}(\mathtt{flat}(x), v_0) = \mathtt{aflat}(x, v_0) \land \tag{4}$$

$$\forall v_1.\mathtt{app}(\mathtt{flat}(z), v_1) = \mathtt{aflat}(z, v_1) \;\to \tag{5}$$

$$\mathtt{app}(\mathtt{flat}(\mathtt{node}(x, y, z)), v) = \mathtt{aflat}(\mathtt{node}(x, y, z), v)\big) \tag{6}$$

For proving (6), we first use the recursive definitions at lines 14 and 16 of Figure 1 to obtain (omitting (4), (5) and implicit universal quantification):

$$\begin{aligned}\mathtt{app}(\mathtt{app}(\mathtt{flat}(x), \mathtt{cons}(y, \mathtt{flat}(z))), v) = \\ \mathtt{aflat}(x, \mathtt{cons}(y, \mathtt{aflat}(z, v)))\end{aligned} \tag{7}$$

By rewriting (7) with (4) and (5), we are left with proving:

$$\begin{aligned}\mathtt{app}(\mathtt{app}(\mathtt{flat}(x), \mathtt{cons}(y, \mathtt{flat}(z))), v) = \\ \mathtt{app}(\mathtt{flat}(x), \mathtt{cons}(y, \mathtt{app}(\mathtt{flat}(z), v)))\end{aligned} \tag{8}$$

By replacing $\mathtt{flat}(x)$ with a fresh variable w in (8), we obtain

$$\begin{aligned}\mathtt{app}(\mathtt{app}(w, \mathtt{cons}(y, \mathtt{flat}(z))), v) = \\ \mathtt{app}(w, \mathtt{cons}(y, \mathtt{app}(\mathtt{flat}(z), v)))\end{aligned} \tag{9}$$

which is a generalized/stronger formula than (8). By applying the structural induction formula over lists

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

over w in (9), we derive the validity of (9) by also using the definition of app from lines 11-12 in Figure 1. We thus conclude that (1) holds, and hence the assertion at line 17 of Figure 1 is valid.

While the proof above is quite natural for humans, it is very difficult for saturation-based first-order provers using the superposition calculus. For example, the state-of-the-art solvers supporting induction CVC4 [15], ZIPPERPOSITION [16] and VAMPIRE [17] fail proving (1). To organize proof search, saturation-based theorem provers, intuitively speaking, disallow rewriting small terms into big terms w.r.t. some ordering. In most (simplification) orderings used by these provers, the terms $\mathtt{flat}$ and $\mathtt{aflat}$ in (6) cannot be expanded using their recursive definitions, as the right-hand sides of these definitions are heavier/bigger[1] than their left-hand sides. Moreover, deciding the order in which induction hypotheses should be applied, such as (4) and (5), is as difficult as doing the proof itself. In this paper, *we extend superposition reasoning with special treatment of recursive definitions, guiding the generation of induction formulas during saturation (Section IV). We use rewrite rules for terms occurring in recursive definitions and inductive hypotheses (Section V).* Thanks to this extension, our work can easily validate (1). $\square$

Another challenging aspect of induction with recursive definitions comes with generalizing and adjusting induction formulas over recursively defined terms and multiple premises, as illustrated next.

Example 2 (Inductive reasoning with naturals)**.** Using the recursive function and predicate definitions of $\mathtt{add}$, $\mathtt{mul}$, and $\mathtt{even}$ from lines 4–10 of Figure 1, the assertion at line 18 encodes the following first-order formula over naturals:

$$\forall x, y.\mathtt{even}(y) \rightarrow \mathtt{even}(\mathtt{mul}(x, y)) \tag{10}$$

Similarly as in Example 1, proving (10) requires instantiating a structural induction formula for naturals as below:

$$\big(F[\mathtt{zero}] \land \forall z.(F[z] \rightarrow F[\mathtt{s}(z)])\big) \rightarrow \forall x.F[x] \tag{11}$$

and thereby proving the following two formulas:

$$\forall y.\mathtt{even}(y) \rightarrow \mathtt{even}(\mathtt{mul}(\mathtt{zero}, y)) \tag{12}$$

$$\begin{aligned}\forall z, y.\big(\big(\mathtt{even}(y) \rightarrow \mathtt{even}(\mathtt{mul}(z, y))\big) \rightarrow \\ \big(\mathtt{even}(y) \rightarrow \mathtt{even}(\mathtt{mul}(\mathtt{s}(z), y))\big)\big)\end{aligned} \tag{13}$$

Validity of the formula (12) follows from the recursive function definitions in lines 6 and 8 of Figure 1. By using the recursive definition in line 7 of Figure 1, formula (13) reduces to

$$\forall z, y.\big(\mathtt{even}(\mathtt{mul}(z, y)) \rightarrow \mathtt{even}(\mathtt{add}(\mathtt{mul}(z, y), y))\big) \tag{14}$$

The antecedent of (14) cannot however be used for proving its conclusion. We overcome this limitation by replacing/generalizing $\mathtt{mul}(z, y)$ in (14) with a fresh new variable u and instantiating the following variant of (11):

$$\begin{aligned}&\big(F[\mathtt{zero}] \land F[\mathtt{s}(\mathtt{zero})] \land \forall z.(F[z] \rightarrow F[\mathtt{s}(\mathtt{s}(z))])\big) \\ &\rightarrow \forall x.F[x]\end{aligned} \tag{15}$$

While (11) cannot be used to prove (14), note that (15) enables the application of the recursive definition of $\mathtt{even}$ in line 10 of Figure 1. As such, proving the generalized version of (14) reduces to proving the three formulas:

$$\mathtt{even}(\mathtt{zero}) \rightarrow \mathtt{even}(\mathtt{add}(\mathtt{zero}, y)) \tag{16}$$

$$\mathtt{even}(\mathtt{s}(\mathtt{zero})) \rightarrow \mathtt{even}(\mathtt{add}(\mathtt{s}(\mathtt{zero}), y)) \tag{17}$$

$$\begin{aligned}\forall z.\big(\big(\mathtt{even}(z) \rightarrow \mathtt{even}(\mathtt{add}(z, y))\big) \rightarrow \\ \big(\mathtt{even}(\mathtt{s}(\mathtt{s}(z))) \rightarrow \mathtt{even}(\mathtt{add}(\mathtt{s}(\mathtt{s}(z)), y))\big)\big)\end{aligned} \tag{18}$$

All three formulas can be proven by applying the recursive function definitions of $\mathtt{add}$ and $\mathtt{even}$ from Figure 1 and using induction with multiple premises over (18) (Section VI). In this paper, *we generate induction formula variants, such as* (15)*, based on recursive function/predicate definitions (Section IV) and support induction with multiple premises (Section VI)*, proving for example (10). $\square$

While relatively simple, Figure 1 illustrates the key challenges in automating induction with recursive definitions in superposition: (i) *strengthening and creating induction formulas* using recursive definitions (Section IV); (ii) *rewriting recursively defined terms* by their (function/predicate) definitions (Section V); and (iii) *applying induction with multiple premises* (Section VI). In what follows, we describe our solutions for these challenges.

III. PRELIMINARIES

We assume familiarity with *standard multi-sorted first-order logic with equality*. Functions are denoted with f, g, h, predicates with p, q, r, variables with x, y, z, u, v, w, and Skolem constants with σ, all possibly with indices. A term is *ground* if it contains no variables. By $\overline{x}$ and $\overline{t}$ we denote tuples of variables and terms, respectively.

We use the standard logical connectives $\neg$, $\lor$, $\land$, $\rightarrow$ and $\leftrightarrow$, and quantifiers $\forall$ and $\exists$. A *literal* is an atom or its negation. For a literal L, we write $\overline{L}$ to denote its complementary literal. A disjunction of literals is a *clause*. We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. We denote the *clausal normal form* of a formula F by $\mathtt{cnf}(F)$. 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 use the words *sort* and *type* interchangeably. We distinguish special sorts called *inductive sorts*, function symbols for inductive sorts called *constructors* and *destructors*. We distinguish *recursive constructors*, which have at least one argument of the same sort as their return sort, from *base constructors*, which do not have any arguments of the same type as their return sort. We call the ground terms built from the constructor symbols of a sort its *term algebra*.

We axiomatise term algebras using their *injectivity*, *distinctness*, *exhaustiveness* and *acyclicity* axioms [14]. In this paper, we refer to term algebras also as algebraic data types or inductively defined data types.

We write $E[s]$ to denote that expression E contains k distinguished occurrence(s) of the term s, with $k \geq 0$. For simplicity, $E[t]$ means that these occurrences of s are replaced by the term t. Further, $E[t]_{p_1 \dots p_k}$, with $p_1 \dots p_k \in \{0, 1\}^k$,

[1]W.r.t. orderings of first-order provers.

is the expression obtained by replacing ith distinguished occurrence of s by t in $E[s]$ iff $p_i = 1$. We abbreviate $E[t_1]\dots[t_n]$ with $E[\overline{t}]$.

A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two terms s and t if $s\theta = t\theta$, and is a *most general unifier* (*mgu*) if for every unifier η of s and t, there exists substitution μ s.t. $\eta = \theta\mu$. We denote the mgu of s and t with $\mathtt{mgu}(s,t)$.

A. Saturation-based proof search

First-order theorem provers work with clauses, rather than with arbitrary formulas. Given a set S of input clauses, first-order provers *saturate* S by computing all logical consequences of S with respect to a sound inference system $\mathcal{I}$. The saturated set of S is called the *closure* of S and process of computing the closure of S is called *saturation* [22]. If the closure contains the empty clause $\Box$, the original set S of clauses is unsatisfiable. A simplified saturation algorithm for inference system $\mathcal{I}$ is given below with a clausified goal F and clausified assumptions A as input:

```
passive := $A \cup \{\neg F\}$, active := $\emptyset$
while passive $\neq \emptyset$:
   $G$ := select(passive)
   derive consequences $\mathcal{C}$ of $G$ and active w.r.t. $\mathcal{I}$
   passive := (passive $\cup\, \mathcal{C}) \setminus G$
   active := active $\cup \{G\}$
   if $\Box \in$ passive then return UNSAT
return SAT
```

Completeness and efficiency of saturation-based reasoning rely heavily on properties of *select* and $\mathcal{I}$ (lines 3 and 4). The *superposition calculus* [19] (denoted $\mathbb{S}\mathrm{up}$) is the most common inference system employed by saturation-based first-order theorem provers, such as E [20], VAMPIRE [22] and ZIPPERPOSITION [16]. The superposition calculus is *sound* and *refutationally complete*: for any unsatisfiable formula, the empty clause can be derived as a logical consequence. To organize saturation, first-order provers use simplification *orderings* on terms, which are extended to orderings over literals and clauses; for simplicity, we write $\succ$ for both the term ordering and its clause ordering extension. We write $s \doteq t$ to mean that the orientation of the equality $s = t$ is fixed (i.e., either $s \succ t$ or $t \succ s$).

We make use of the following inference rules of $\mathbb{S}\mathrm{up}$ in this paper:

Binary resolution:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\theta}$$

where θ is the mgu of A and B.

Superposition:

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

where θ is the mgu of l and l', $r\theta \not\succeq l\theta$ and $t\theta \not\succeq s[l']\theta$. There are special cases of these rules, imposing more restrictions on the premises. One such case is when one of the premises of superposition is a unit clause, yielding the so-called *demodulation* rules, as given in Section V.

Given an ordering $\succ$, a clause C is *redundant* with respect to a set S of clauses if there exists a subset S' of S such that S' is smaller than $\{C\}$ (i.e., $C \succ S$) and S' implies C. Redundant clauses can be eliminated during proof search without destroying completeness; *simplification and deletion rules* are used to remove redundant clauses.

IV. Induction Formulas over Recursive Definitions in Superposition

We now describe our solution for generating induction formulas in saturation-based theorem proving. Unlike [7], [4], [16], [10], [11], [25], [26], we integrate induction directly in the saturation-based theorem proving using the superposition calculus. For doing so, we rely on [17], [18] and use the following sound *inference rule of induction*:

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

where L is a ground literal, C is a clause, and $F \rightarrow \forall \overline{y}.L[\overline{y}]$ is a valid induction formula. Further, $\overline{y}$ is a tuple of variables and $\overline{t}$ is a tuple of induction terms, of the same size.

In [17], [18], the inference rule $(\mathtt{Ind})$ has been used by considering the induction formulas as instances of mathematical and structural induction. In this paper, we go beyond these works and utilise recursive function/predicate definitions to derive induction formulas to be used in $(\mathtt{Ind})$. For doing so, we first select terms in recursive definitions over which induction formulas will be generated in Section IV-A and strengthened in Section IV-B. Further, in Section VI we extend $(\mathtt{Ind})$ to induction formulas with multiple premises.

A. Generating Induction Formulas over Recursive Definitions

A recursive function/predicate definition has a number of branches, characterized by one or more clauses. We assume that (i) a function definition clause contains exactly one equality with a fixed orientation, i.e., $\mathtt{f}(\overline{s}) \doteq t \vee C$. Similarly, (ii) a predicate definition axiom contains one marked literal, i.e., $(\neg)\hat{\mathtt{p}}(\overline{s}) \vee D$, where $\hat{\mathtt{p}}$ denotes that $\mathtt{p}$ is marked/selected. Two clauses $\mathtt{f}(\overline{s_1}) \doteq t_1 \vee C$ and $\mathtt{f}(\overline{s_2}) \doteq t_2 \vee D$ belong to the same branch of $\mathtt{f}$ if $\mathtt{f}(\overline{s_1})$ and $\mathtt{f}(\overline{s_2})$ are variants of each other. Similarly, two clauses $(\neg)\hat{\mathtt{p}}(\overline{s_1}) \vee C$ and $(\neg)\hat{\mathtt{p}}(\overline{s_2}) \vee D$ belong to the same branch of $\mathtt{p}$ if $\mathtt{p}(\overline{s_1})$ and $\mathtt{p}(\overline{s_2})$ are variants of each other. We therefore characterize a recursive definition branch with its *characteristic term* $\mathtt{f}(\overline{s})$ or *characteristic atom* $\mathtt{p}(\overline{s})$. We write "branch $\mathtt{f}(\overline{s})$" and "branch $\mathtt{p}(\overline{s})$" to refer to the branches with the characteristic term $\mathtt{f}(\overline{s})$ and characteristic atom $\mathtt{p}(\overline{s})$, respectively. We denote the set of variable disjoint branches of a function $\mathtt{f}$ and predicate $\mathtt{p}$ with $\mathcal{B}_{\mathtt{f}}$ and $\mathcal{B}_{\mathtt{p}}$, respectively.

Definition 1 (Recursive Calls of Recursive Definitions)**.** Let $\mathtt{f}$ be a recursive function and $\mathtt{p}$ a recursive predicate. The *set of* *recursive calls* corresponding, respectively, to the branch $\mathtt{f}(\overline{s})$ and the branch $\mathtt{p}(\overline{s})$ are defined as:

$$\mathcal{R}_{\mathtt{f}(\overline{s})} := \bigcup_{\mathtt{f}(\overline{s'}) \dot{=} t \vee C} \{\mathtt{f}(\overline{s''})\theta \mid \mathtt{f}(\overline{s''}) \trianglelefteq t, \mathtt{f}(\overline{s'})\theta = \mathtt{f}(\overline{s})\}$$

$$\mathcal{R}_{\mathtt{p}(\overline{s})} := \bigcup_{\hat{\mathtt{p}}(\overline{s'}) \vee C} \{\mathtt{p}(\overline{s''})\theta \mid \mathtt{p}(\overline{s''}) \in C, \mathtt{p}(\overline{s'})\theta = \mathtt{p}(\overline{s})\}$$

□

The rest of this section only details the generation of induction formulas using recursive function definitions; recursive predicates are handled similarly. Given a recursive function $\mathtt{f}$, we categorize its argument positions similarly to [16].

Definition 2 (Active Positions, Accumulators)**.** If for any branch $\mathtt{f}(\overline{s}) \in \mathcal{B}_\mathtt{f}$ and $\mathtt{f}(\overline{s'}) \in \mathcal{R}_{\mathtt{f}(\overline{s})}$:

(1) if $s'_i \triangleleft s_i$, then i is an *active* argument position of $\mathtt{f}$

(2) if s_i is a variable and $s_i \neq s'_i$, then i is an *accumulator* argument position of $\mathtt{f}$

We denote the set of active and accumulator argument positions of $\mathtt{f}$ with $I_\mathtt{f}$. □

Example 3. Based on the functions $\mathtt{app}$, $\mathtt{flat}$ and $\mathtt{aflat}$ from Figure 1 lines 11-16, we have:

$$\mathcal{B}_{\mathtt{app}} = \{\mathtt{app}(\mathtt{nil}, z_0),\ \mathtt{app}(\mathtt{cons}(x, y), z_1)\}$$

$$\mathcal{B}_{\mathtt{flat}} = \{\mathtt{flat}(\mathtt{leaf}),\ \mathtt{flat}(\mathtt{node}(x, y, z))\}$$

$$\mathcal{B}_{\mathtt{aflat}} = \{\mathtt{aflat}(\mathtt{leaf}, u_0),\ \mathtt{aflat}(\mathtt{node}(x, y, z), u_1)\}$$

While $\mathcal{R}_{\mathtt{app}(\mathtt{nil},z_0)} = \mathcal{R}_{\mathtt{flat}(\mathtt{leaf})} = \mathcal{R}_{\mathtt{aflat}(\mathtt{leaf},u_0)} = \emptyset$, the second branches of the three functions have the following sets of recursive calls:

$$\mathcal{R}_{\mathtt{app}(\mathtt{cons}(x,y),z_1)} = \big\{\mathtt{app}(y, z_1)\big\}$$
$$\mathcal{R}_{\mathtt{flat}(\mathtt{node}(x,y,z))} = \big\{\mathtt{flat}(x),\ \mathtt{flat}(z)\big\}$$
$$\mathcal{R}_{\mathtt{aflat}(\mathtt{node}(x,y,z),u_1)} = \left\{\begin{matrix}\mathtt{aflat}(x, \mathtt{cons}(y, \mathtt{aflat}(z, u_1))),\\ \mathtt{aflat}(z, u_1)\end{matrix}\right\}$$

$I_{\mathtt{app}} = \{1\}$, since y is a proper subterm of $\mathtt{cons}(x, y)$ but the second argument is not an accumulator since it remains z_1 in the only recursive call. The only argument position of $\mathtt{flat}$ is active, and therefore $I_{\mathtt{flat}} = \{1\}$. Finally, $\mathtt{aflat}$ has one active and one accumulator argument position, hence $I_{\mathtt{aflat}} = \{1, 2\}$. □

Definition 3 (Induction Terms from Active and Accumulator Positions)**.** Consider a recursive function $\mathtt{f}$ of arity n and a ground term $\mathtt{f}(\overline{c})$. The term $\mathtt{f}(\overline{c'})$ is a *generator term* iff (i) $\overline{c'}$ coincides with $\overline{c}$ in all positions from $\{1 \leq i \leq n\} \setminus I_\mathtt{f}$, and (ii) $\overline{c'}$ contains fresh variables on positions from $I_\mathtt{f}$.

The *induction case* of $\mathtt{f}(\overline{c})$ over branch $\mathtt{f}(\overline{s}) \in \mathcal{B}_\mathtt{f}$ is the two-tuple:

$$(\theta, \{\mathtt{mgu}(\mathtt{f}(\overline{c'}), \mathtt{f}(\overline{s'})\theta) \mid \mathtt{f}(\overline{s'}) \in \mathcal{R}_{\mathtt{f}(\overline{s})}\})$$

where $\theta := \mathtt{mgu}(\mathtt{f}(\overline{c'}), \mathtt{f}(\overline{s}))$.

The *case distinction* $\Theta_{\mathtt{f}(\overline{c})}$ of $\mathtt{f}(\overline{c})$ is the set of induction cases of $\mathtt{f}(\overline{c})$ over each branch of $\mathtt{f}$. We call $\{c_i \mid i \in I_\mathtt{f}\}$ the *induction terms* of $\mathtt{f}(\overline{c})$. □

Induction Formula over Active and Accumulator Terms. Using Definition 3, we guide induction formula generation over active and accumulator terms, as follows. Given a literal $L[\overline{c}]$ with zero or more occurrences of the terms $\overline{c}$, we generate and add the following *induction formula over active and accumulator terms* to saturation-based proving:

$$(\forall) \bigwedge_{(\theta,R)\in\Theta_{\mathtt{f}(\overline{c})}} \Big(\bigwedge_{\theta' \in R} L[\overline{c'}]\theta' \rightarrow L[\overline{c'}]\theta\Big) \rightarrow L[\overline{c'}] \qquad (19)$$

Since (19) is a valid induction formula, using it in the conclusion of $(\mathtt{Ind})$ yields a sound $(\mathtt{Ind})$ inference.

Example 4. For proving the assertion of line 17 from Figure 1 in a saturation-based framework, we consider its negation:

$$\mathtt{app}(\mathtt{flat}(\sigma_0), \sigma_1) \neq \mathtt{aflat}(\sigma_0, \sigma_1) \qquad (20)$$

Using Definition 3 and $I_{\mathtt{flat}}$ (Example 3), the generator term of $\mathtt{flat}(\sigma_0)$ is $t := \mathtt{flat}(v)$. Moreover, by $\mathcal{B}_{\mathtt{flat}}$ from Example 3, we obtain

$$\theta_1 = \mathtt{mgu}(t, \mathtt{flat}(\mathtt{leaf})) = \{v \mapsto \mathtt{leaf}\}$$
$$\theta_2 = \mathtt{mgu}(t, \mathtt{flat}(\mathtt{node}(x, y, z))) = \{v \mapsto \mathtt{node}(x, y, z)\}$$

Applying the unifier θ_2 on the recursive calls of $\mathcal{R}_{\mathtt{flat}(\mathtt{node}(x,y,z))}$ from Example 3 is a no-op, since the recursive calls do not contain v and we derive

$$\theta_{2.1} = \mathtt{mgu}(t, \mathtt{flat}(x)) = \{v \mapsto x\}$$
$$\theta_{2.2} = \mathtt{mgu}(t, \mathtt{flat}(z)) = \{v \mapsto z\}$$

Using the case distinction

$$\Theta_{\mathtt{flat}(\sigma_0)} = \{(\theta_1, \emptyset),\ (\theta_2, \{\theta_{2.1}, \theta_{2.2}\})\} \qquad (21)$$

we derive the following induction formula:

$$\begin{aligned}
&\forall x, y, z, u.\\
&\Big(\big(\mathtt{app}(\mathtt{flat}(\mathtt{leaf}), \sigma_1) = \mathtt{aflat}(\mathtt{leaf}, \sigma_1) \wedge\\
&\ \big(\mathtt{app}(\mathtt{flat}(x), \sigma_1) = \mathtt{aflat}(x, \sigma_1) \wedge\\
&\ \ \mathtt{app}(\mathtt{flat}(z), \sigma_1) = \mathtt{aflat}(z, \sigma_1) \ \rightarrow\\
&\ \ \mathtt{app}(\mathtt{flat}(\mathtt{node}(x, y, z)), \sigma_1) = \mathtt{aflat}(\mathtt{node}(x, y, z), \sigma_1)\big)\big)\\
&\ \rightarrow \mathtt{app}(\mathtt{flat}(u), \sigma_1) = \mathtt{aflat}(u, \sigma_1)\Big)
\end{aligned} \qquad (22)$$

□

B. Strengthening Induction over Recursive Definitions

Induction hypotheses of induction formulas might not be strong enough to prove the corresponding induction step. A common technique to overcome such limitations is to strengthen the induction hypotheses: replace some terms in the hypotheses with universally quantified fresh variables, yielding thus logically stronger versions of induction hypotheses. Introducing universally quantified variables during saturation can however negatively impact the performance of the prover (e.g., yielding more unifications/rewriting steps). As a remedy to this practical burden in the context of recursive function definitions $\mathtt{f}$, we utilize the *accumulator argument positions* from $I_\mathtt{f}$ in Definition 3, which supersede the need for introducing universally quantified variables by implicitly instantiating these variables to the terms that will be matched by the recursive calls of $\mathtt{f}$.

Example 5. The induction formula (22) is not strong enough to prove (20) and strengthening its induction hypotheses by replacing σ_1 with a universally quantified fresh variable – as in (4) and (5) from Example 1, – is inefficient. Instead, we use the term $\mathtt{aflat}(\sigma_0, \sigma_1)$ from (20) with the generator term $t' := \mathtt{aflat}(v, w)$ and induction terms $\{\sigma_0, \sigma_1\}$. We obtain the following unifiers:

$$\begin{aligned}\theta'_1 &= \mathtt{mgu}(t', \mathtt{aflat}(\mathtt{leaf}, u_0)) = \{v \mapsto \mathtt{leaf}, w \mapsto u_0\}\\ \theta'_2 &= \mathtt{mgu}(t', \mathtt{aflat}(\mathtt{node}(x, y, z), u_1))\\ &= \{v \mapsto \mathtt{node}(x, y, z), w \mapsto u_1\}\end{aligned}$$

Applying θ'_2 is once again a no-op on the recursive calls $\mathcal{R}_{\mathtt{aflat}(\mathtt{node}(x,y,z),u_1)}$, and we get the unifiers:

$$\begin{aligned}\theta'_{2.1} &= \mathtt{mgu}(t', \mathtt{aflat}(x, \mathtt{cons}(y, \mathtt{aflat}(z, u_1))))\\ &= \{v \mapsto x, w \mapsto \mathtt{cons}(y, \mathtt{aflat}(z, u_1))\}\\ \theta'_{2.2} &= \mathtt{mgu}(t', \mathtt{aflat}(z, u_1)) = \{v \mapsto z, w \mapsto u_1\}\end{aligned}$$

Thus we obtain the induction formula with the required induction hypothesis with term $\mathtt{cons}(y, \mathtt{aflat}(z, u_1))$ that matches the conclusion after simplification:

$$\begin{aligned}&\forall x, y, z, u_0, u_1, v, w.\\ &\Big(\big(\mathtt{app}(\mathtt{flat}(\mathtt{leaf}), u_0) = \mathtt{aflat}(\mathtt{leaf}, u_0) \wedge\\ &\big(\mathtt{app}(\mathtt{flat}(x), \mathtt{cons}(y, \mathtt{aflat}(z, u_1))) =\\ &\qquad \mathtt{aflat}(x, \mathtt{cons}(y, \mathtt{aflat}(z, u_1))) \wedge\\ &\quad \mathtt{app}(\mathtt{flat}(z), u_1) = \mathtt{aflat}(z, u_1) \rightarrow\\ &\quad \mathtt{app}(\mathtt{flat}(\mathtt{node}(x, y, z)), u_1) = \mathtt{aflat}(\mathtt{node}(x, y, z), u_1)\big)\big)\\ &\quad \rightarrow \mathtt{app}(\mathtt{flat}(v), w) = \mathtt{aflat}(v, w)\Big)\end{aligned} \tag{23}$$

After skolemizing x, y, z, u_0 and u_1 during clausification, binary resolving with (20), with v and w bound to σ_0 and σ_1, respectively, we get the following ground induction hypotheses literals and ground conclusion literal from (23):

$$\begin{aligned}&\mathtt{app}(\mathtt{flat}(\sigma_2), \mathtt{cons}(\sigma_3, \mathtt{aflat}(\sigma_4, \sigma_5))) =\\ &\qquad \mathtt{aflat}(\sigma_2, \mathtt{cons}(\sigma_3, \mathtt{aflat}(\sigma_4, \sigma_5)))\end{aligned} \tag{24}$$

$$\mathtt{app}(\mathtt{flat}(\sigma_4), \sigma_5) = \mathtt{aflat}(\sigma_4, \sigma_5) \tag{25}$$

$$\begin{aligned}&\mathtt{app}(\mathtt{flat}(\mathtt{node}(\sigma_2, \sigma_3, \sigma_4)), \sigma_5) \neq\\ &\qquad \mathtt{aflat}(\mathtt{node}(\sigma_2, \sigma_3, \sigma_4), \sigma_5)\end{aligned} \tag{26}$$

Further, the hypotheses of (23) are strong enough to prove (20), as shown in Section V. □

In summary, we use Definition (3) to generate induction formulas over the active and accumulator terms from $I_{\mathtt{f}}$. To further limit and guide the generation of induction formulas, we devised *heuristics* similar to [16]. Foremost, we only generate induction formulas from function/predicate terms with active occurrences.

Definition 4 (Active Term Occurrences)**.** An occurrence of a term t in literal L is an *active occurrence* if (i) t is L, or (ii) L is an equality $l = r$ and t is l or r, or (iii) the immediate superterm s of t is an active occurrence and the occurrence of t is in an active argument position of s. □

As described in [18], apart from generalizing over complex terms as seen in Example (1), we can also generalize over active term occurrences. For example, we can refine the induction formula (19) to induct upon only certain occurrences of an induction term t with k occurrences in literal L, by using any bit vector $p \in \{0, 1\}^k$ and $L[t]_p$ instead of $L[t]$.

V. Refuting Inductive Properties with Recursive Definitions

Automating inductive reasoning not only requires finding useful induction formulas, but also comes with the task of proving inductive properties. Section IV detailed our approach towards finding useful induction formulas over recursive definitions. As a next step, we now present our solution towards (more) efficient refutation of inductive properties over recursive definitions.

A. Rewriting with Recursive Function Definitions

We extend superposition reasoning with two inference rules in support of rewriting recursive functions by their definitions.

First, we focus on a *simplification inference* implementing rewriting by unit equalities, called also demodulation [22]. We adjust demodulation to handle unit clauses describing recursive function definitions, as follows:

$$\frac{f(\overline{s}) \doteq t \quad \cancel{L[f(\overline{s})\theta] \vee D}}{L[t\theta] \vee D}\ (\mathtt{DemF})$$

where $f(\overline{s})\theta \succ t\theta$ and $L[f(\overline{s})\theta] \vee D \succ f(\overline{s})\theta = t\theta$.

Second, we introduce a *generating inference* rule as an instance of superposition rules. Namely, we enable rewriting arbitrary recursive functions with their definitions, as follows:

$$\frac{f(\overline{s}) \doteq t \vee C \quad L[f(\overline{s})\theta] \vee D}{L[t\theta] \vee C\theta \vee D}\ (\mathtt{ParF})$$

Note that $(\mathtt{ParF})$ has no side conditions restricting which terms can be rewritten. As such, $(\mathtt{ParF})$ allows to expand function headers, yet at the cost that small terms may be rewritten into bigger terms w.r.t. the underlining term ordering $\succ$ of a superposition prover. As a result, the simplification ordering constraints of $\succ$ are violated by $(\mathtt{ParF})$, yielding an incomplete extension of superposition. On the other hand, soundness of superposition implies soundness of our new inference rules.

Theorem 1 (Soundness of Rewriting)**.** The inference rules $(\mathtt{DemF})$ and $(\mathtt{ParF})$ are sound. □

B. Rewriting Induction Hypotheses

Upon clausifying the induction formula (19) introduced in Section IV, for each step case $\wedge_{1\leq i\leq m} L[\overline{t_i}] \rightarrow L[\overline{t}]$ we obtain a set of *induction hypothesis literals* $L[\overline{t'_i}]$ and an *induction conclusion literal* $\overline{L[\overline{t'}]}$. Intuitively, we extend these notions such that any literal resulting from the rewriting or simplification of induction hypothesis or induction conclusion literals is also an induction hypothesis or induction conclusion literal, respectively.

We introduce an *induction hypothesis rewriting rule*, in short $(\mathtt{IndHRW})$, to (i) rewrite one side of an induction conclusion literal with one of its induction hypothesis literals (against ordering constraints) and (ii) apply induction on the rewritten induction conclusion literal without adding it to the search space:

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

where $s \neq t$ is an induction conclusion literal with corresponding induction hypothesis literal $l = r$, $l \not\succeq r$, and $F \rightarrow \forall \overline{y}.(s[r] = t)[\overline{y}]$ is a valid induction formula. By soundness of $(\texttt{Ind})$, we conclude soundness of $(\texttt{IndHRW})$.

Theorem 2 (Soundness of Induction Hypothesis Rewriting)**.** The inference rule $(\texttt{IndHRW})$ is sound. □

Note that $(\texttt{IndHRW})$ allows rewriting only with induction hypothesis literals that are positive equalities. Hence, the induction conclusion literal must be a disequality ($s \neq t$). We further stress that rewriting using the premises of $(\texttt{IndHRW})$ yields $s[r] \neq t \vee C \vee D$, which is binary resolved against the resulting induction formula clauses of (19) and not added to the search space.

Example 6. Continuing Example 5, rewriting (26) with $(\texttt{ParF})$ results in a new induction conclusion literal:

$$\begin{aligned}&\texttt{app}(\texttt{app}(\texttt{flat}(\sigma_2), \texttt{cons}(\sigma_3, \texttt{flat}(\sigma_4))), \sigma_5) \neq \\ &\qquad \texttt{aflat}(\sigma_2, \texttt{cons}(\sigma_3, \texttt{aflat}(\sigma_4, \sigma_5)))\end{aligned} \tag{27}$$

By rewriting the right-hand side of (27) with the corresponding hypotheses literals (24) and (25), we obtain the intermediate induction conclusion literal

$$\begin{aligned}&\texttt{app}(\texttt{app}(\texttt{flat}(\sigma_2), \texttt{cons}(\sigma_3, \texttt{flat}(\sigma_4))), \sigma_5) \neq \\ &\qquad \texttt{app}(\texttt{flat}(\sigma_2), \texttt{cons}(\sigma_3, \texttt{app}(\texttt{flat}(\sigma_4), \sigma_5)))\end{aligned} \tag{28}$$

By applying induction with $(\texttt{IndHRW})$ with case distinction $\Theta_{\texttt{app}(\texttt{flat}(\sigma_2), \texttt{cons}(\sigma_3, \texttt{flat}(\sigma_4)))}$ and induction term $\texttt{flat}(\sigma_2)$, we obtain the induction formula:

$$\begin{aligned}&\forall x, y, z. \\ &\Big(\big(\texttt{app}(\texttt{app}(\texttt{nil}, \texttt{cons}(\sigma_3, \texttt{flat}(\sigma_4))), \sigma_5) = \\ &\qquad \texttt{app}(\texttt{nil}, \texttt{cons}(\sigma_3, \texttt{app}(\texttt{flat}(\sigma_4), \sigma_5))) \wedge \\ &\ \big(\texttt{app}(\texttt{app}(y, \texttt{cons}(\sigma_3, \texttt{flat}(\sigma_4))), \sigma_5) = \\ &\qquad \texttt{app}(y, \texttt{cons}(\sigma_3, \texttt{app}(\texttt{flat}(\sigma_4), \sigma_5))) \rightarrow \\ &\ \ \texttt{app}(\texttt{app}(\texttt{cons}(x, y), \texttt{cons}(\sigma_3, \texttt{flat}(\sigma_4))), \sigma_5) = \\ &\qquad \texttt{app}(\texttt{cons}(x, y), \texttt{cons}(\sigma_3, \texttt{app}(\texttt{flat}(\sigma_4), \sigma_5)))\big)\big) \\ &\ \rightarrow \texttt{app}(\texttt{app}(z, \texttt{cons}(\sigma_3, \texttt{flat}(\sigma_4))), \sigma_5) = \\ &\qquad \texttt{app}(z, \texttt{cons}(\sigma_3, \texttt{app}(\texttt{flat}(\sigma_4), \sigma_5)))\Big)\end{aligned} \tag{29}$$

The resulting clauses – after binary resolving with the intermediate unit clause (28) – can be finally refuted using the definitions at lines 11 and 12 of Figure 1. We thus validate correctness of the assertion on line 17 in Figure 1. □

VI. Multi-Clause Induction in Superposition

The induction rule $(\texttt{Ind})$ does not allow inducting on multiple literals, limiting for example the use of $(\texttt{Ind})$ over (14) in Example 2. Moreover, when $(\texttt{Ind})$ is used together with the induction formula (19), clausification introduces new Skolem constants, making it impossible to use ground assumptions or previous induction hypotheses containing different ground subterms. To address this issue, in this section we revise the induction inference rule $(\texttt{Ind})$ with only one premise to an *induction rule with multiple premises*, as follows.

We extend $(\texttt{Ind})$ for a given literal $\overline{L}$ (the *main literal*) to also incorporate other literals L_i (the *side literals*) that are relevant for proving $\overline{L}$, as follows:

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

where $\overline{L}$ and L_i are ground literals, C and C_i are clauses, and $F \rightarrow \forall \overline{y}.(\bigwedge_{1 \leq i \leq n} L_i[\overline{y}] \rightarrow L[\overline{y}])$ is a valid induction formula. Further, $\overline{y}$ and $\overline{t}$ are tuples of variables and induction terms, respectively. Soundness of $(\texttt{IndMC})$ follows then from soundness of $(\texttt{Ind})$.

Theorem 3 (Soundness of Multi-clause Induction)**.** The rule $(\texttt{IndMC})$ is sound. □

We note that after the application of $(\texttt{IndMC})$, binary resolution can be applied on each resulting clause with the main and side literals, yielding $\texttt{cnf}(\neg F) \vee \bigvee_{1 \leq i \leq n} C_i \vee C$.

Multi-Clause Induction Formula over Active and Accumulator Terms. For generating valid induction formulas to be used in $(\texttt{IndMC})$, we proceed as in Section IV. Yet, we adjust the generation of (19), by using Definition 3 over the active and accumulator terms of $\wedge_{k=1}^{n} L_k[\overline{c'}] \rightarrow L[\overline{c'}]$ (rather than just $L[\overline{c}]$). As a result, for a given case distinction $\Theta_{\texttt{f}(\overline{c})}$, we generate the following *multi-clause induction formula over active and accumulator terms* in saturation-based proving:

$$\begin{aligned}&(\forall) \bigwedge_{(\theta, R) \in \Theta_{\texttt{f}(\overline{c})}} \Big(\bigwedge_{\theta' \in R} (\wedge_{k=1}^{n} L_k[\overline{c'}]\theta' \rightarrow L[\overline{c'}]\theta') \rightarrow \\ &\quad (\wedge_{k=1}^{n} L_k[\overline{c'}]\theta \rightarrow L[\overline{c'}]\theta)\Big) \rightarrow (\wedge_{k=1}^{n} L_k[\overline{c'}] \rightarrow L[\overline{c'}])\end{aligned} \tag{30}$$

Since (30) is a valid induction formula, using it in the conclusion of $(\texttt{IndMC})$ yields a sound $(\texttt{IndMC})$ inference.

Example 7. Negating and clausifying the assertion on line 18 of Figure 1, we obtain the two unit clauses:

$$\texttt{even}(\sigma_1) \tag{31}$$

$$\neg\texttt{even}(\texttt{mul}(\sigma_0, \sigma_1)) \tag{32}$$

Inducting on (32) using $\Theta_{\texttt{mul}(\sigma_0, \sigma_1)}$ and induction term σ_0, we get the following clauses:

$$\neg\texttt{even}(\texttt{mul}(\texttt{zero}, \sigma_1)) \vee \texttt{even}(\texttt{mul}(\sigma_2, \sigma_1))$$

$$\neg\texttt{even}(\texttt{mul}(\texttt{zero}, \sigma_1)) \vee \neg\texttt{even}(\texttt{mul}(\texttt{s}(\sigma_2), \sigma_1))$$

By function and predicate definitions of mul and even, the base case reduces to false and we are left with the unit clauses

$$\texttt{even}(\texttt{mul}(\sigma_2, \sigma_1)) \tag{33}$$

$$\neg\texttt{even}(\texttt{add}(\texttt{mul}(\sigma_2, \sigma_1), \sigma_1)) \tag{34}$$

The hypothesis literal in (33) and the conclusion literal in (34) cannot be binary resolved with each other to solve the step case but they share the term $\texttt{mul}(\sigma_2, \sigma_1)$. We can use (33) and (34) in (IndMC) as side and main literals, respectively, with induction term $\mathtt{mul}(\sigma_2, \sigma_1)$ and the case distinction:

$$\Theta_{\mathtt{even}(\mathtt{mul}(\sigma_2,\sigma_1))} = \left\{ \begin{matrix} (\{z \mapsto \mathtt{zero}\}, \emptyset), (\{z \mapsto \mathtt{s}(\mathtt{zero})\}, \emptyset), \\ (\{z \mapsto \mathtt{s}(\mathtt{s}(x))\}, \{\{z \mapsto x\}\}) \end{matrix} \right\}$$

We get the following induction formula:

$$\begin{aligned} \forall x, z. \Big(& \big(\mathtt{even}(\mathtt{zero}) \to \mathtt{even}(\mathtt{add}(\mathtt{zero}, \sigma_1))\big) \land \\ & \big(\mathtt{even}(\mathtt{s}(\mathtt{zero})) \to \mathtt{even}(\mathtt{add}(\mathtt{s}(\mathtt{zero}), \sigma_1))\big) \land \\ & \big(\big(\mathtt{even}(x) \to \mathtt{even}(\mathtt{add}(x, \sigma_1))\big) \to \\ & \quad \big(\mathtt{even}(\mathtt{s}(\mathtt{s}(x))) \to \mathtt{even}(\mathtt{add}(\mathtt{s}(\mathtt{s}(x)), \sigma_1))\big)\big) \\ & \to \big(\mathtt{even}(z) \to \mathtt{even}(\mathtt{add}(z, \sigma_1))\big)\Big) \end{aligned} \tag{35}$$

After clausifying (35), and binary resolving the resulting clauses against (33) and (34), using function and predicate definitions and the unit clause (31), we arrive at the empty clause, thus validating the assertion at line 18 in Figure 1. □

We conclude this section by noting that the (IndMC) inference rule might use an arbitrary number of side literals, slowing down the practical efficiency of saturation-based proving with multi-clause induction. As a remedy, the following two heuristics could be used to choose the literal L from clause $L \lor C$ as a side literal of (IndMC): (i) if L is $\mathtt{p}(\overline{s})$ for some predicate $\mathtt{p}$, and L is an induction hypotheses to the main literal $\overline{\mathtt{p}(\overline{t})}$, and $\overline{s}$ and $\overline{t}$ share some non-Skolem (complex) term with an active occurrence, or (ii) if neither L nor the main literal are derived from a clausified induction formula and they share some common term with an active occurrence.

VII. Experiments

Implementation. We implemented our approach to automating induction with recursive definitions in superposition-based theorem prover Vampire. We extended Vampire's induction framework [18] with recursive definitions and hypothesis strengthening, as described in Section IV. This can be enabled with `--structural_induction_kind rec_def`. Rewriting with induction hypotheses and function definitions, as presented in Section V, can be switched on using `--induction_hypothesis_rewriting on` and `--function_definition_rewriting on`, respectively. The multi-clause induction rule from Section VI is enabled by `--induction_multiclause on`. All together, our implementation consists of around 5,000 lines of C++ code and is available at https://github.com/vprover/vampire/tree/induction-recursive-functions.

Experimental setup. To experimentally evaluate our approach, we used the benchmarking tool BenchExec [27], [28] and two benchmark sets[2]: (i) the UFDTLIA examples from SMT-LIB [24], consisting of 327 problems over algebraic data types; and (ii) our new set dty_RD of 3,397 inductive examples with recursive definitions, as described in [30]. We used the keyword `define-fun-rec` for defining recursive functions in the examples from our dty_RD dataset. Moreover,

[2]While some examples from the TIP library [29] are included in SMT-LIB, most of the TIP examples are parametric and not yet supported by Vampire.

	UFDTLIA 327 problems	dty_RD 3,397 problems
Vampire	180 (0)	1,641 (0)
Vampire*	**259 (30)**	**3,223 (497)**
Zipperposition	174 (0)	2,534 (21)
Cvc4	235 (12)	165 (0)

Fig. 2. Numbers of problems solved by respective solvers in our experiments. The number in parentheses is the number of problems solved uniquely compared to the other solvers.

we also converted examples from the UFDTLIA set to explictly use `define-fun-rec`, detecting this way recursive definitions in UFDTLIA.

We also combined our inductive approach in Vampire with recent developments in first-order reasoning [18], [31], [32], creating this way various Vampire configurations for automating induction with recursive definitions. The *default options* we used for these configurations are: `--induction_gen on --induction_on_complex_terms on` enabling inductive generalizations and induction on complex terms [18]; `--newcnf on` to select the cnf method in [31]; and `--theory_split_queue on --theory_split_queue_cutoffs 0,8` and `--theory_split_queue_ratios 20,10,1` to control theory reasoning with split queues [32]. As a result, we designed a new Vampire portfolio mode for inductive reasoning, which can be switched on by `--mode portfolio --schedule struct_induction`.

Experimental comparison. In what follows, Vampire refers to the (default) version of Vampire, as in [18]. By Vampire* we denote our new version of Vampire, using induction with recursive definitions and the aforementioned options. We compared our work in Vampire* against Vampire, as well as against the superposition prover Zipperposition[3] [16] and the SMT solver Cvc4 [33].

Since the default mode of Vampire and Vampire* only occasionally solves unique problems with respect to their portfolio mode counterpart, we omitted the former results. Note that we used the same portfolio schedule `struct_induction` for Vampire as well. Since in portfolio mode Vampire ignores the new options and most of the schedule is not specific to Vampire*, the results obtained for Vampire give a meaningful baseline. We used Zipperposition in the default mode, while for Cvc4 we used the parameters `--conjecture-gen --quant-ind`. Each prover was given 300 seconds of time and 16 GB of memory per problem. The experiments were ran on computers with 32 cores (AMD Epyc 7502, 2.5 GHz) and 1 TB RAM.

Experimental results. We summarize our experimental results in Figure 2. For each solver, listed in the first column of

[3]Zipperposition has a non-official option `--input tip` to parse benchmarks in a variant of SMT-LIB. In order to parse UFDTLIA benchmarks, we converted them to this variant.

VAMPIRE* forced option	UFDTLIA 327 problems	dty_RD 3,397 problems
default	**259 (1)**	3223 (3)
`-indmc off`	237 (0)	**3259 (33)**
`-indhrw off`	242 (0)	3192 (4)
`-fnrw off`	237 (3)	3001 (0)
`-sik one`	200 (1)	962 (0)

Fig. 3. Numbers of problems solved by VAMPIRE* with different new features disabled. The number in parentheses is the number of problems solved uniquely compared to the other configurations.

Figure 2, we indicate the total number of examples the solver proved from the respective benchmark category; the values in parentheses show the number of uniquely solved problems compared to the other solvers. Figure 2 shows that while VAMPIRE performs reasonably well on both benchmark sets, it cannot solve more problems than CVC4 in the UFDTLIA set and than ZIPPERPOSITION in the dty_RD set, where the latter two perform the best. VAMPIRE*, on the other hand, is able to solve many more problems than the other solvers in both sets, suggesting that combining the state-of-the-art techniques of superposition with induction over recursive definition can perform much better than SMT solvers and superposition provers with only structural induction. All together, **VAMPIRE* solved 527 new problems that the other automated solvers could not prove**. It is also worth noting that while VAMPIRE* dominates the uniquely solved problems w.r.t. the dty_RD set, its dominance is only marginal compared to the uniquely solved problems of CVC4 in the UFDTLIA set. Looking at the problems uniquely solved by CVC4, we found that these problems mostly contain either some nested structure that current techniques in VAMPIRE* cannot handle and require non-trivial lemma generation or recursive definitions that cannot be used with our induction formula generation as their well-foundedness is not based on the subterm relation.

In addition to comparing to other solvers, we compared VAMPIRE* to itself with different techniques from the paper disabled, overriding the portfolio options during these runs. Our results are shown in Figure 3.

For UFDTLIA, the default run still performs best but we can see different deviations from this value with each disabled technique. We argue that the relatively small differences obtained by turning off induction hypothesis rewriting (`-indhrw off`) and function definition rewriting (`-fnrw off`) can be attributed to combinations of options that together may simulate these techniques. In comparison, multi-clause induction cannot be simulated with other techniques in VAMPIRE, so the relatively small difference obtained by turning off this technique (`-indmc off`) for UFDTLIA is probably due to the lack of non-unit induction needed in most of this set. For dty_RD, the decrease in solved problems when this feature is turned on needs further investigation. The greatest difference to the default is obtained by using structural induction (`-sik one`, see [17]) instead of inferring induction formulas from recursive function definitions. We can conclude with the observation that each configuration solved problems uniquely which suggests the portfolio schedule can be improved.

VIII. Related work

Generation of induction formulas, as presented in Section IV, although similar to *recursion analysis* of [7] and *recursion induction* of [10], utilizes unification and generates non-trivial induction hypotheses. Our work complements these techniques by integrating induction in saturation: rather than replacing inductive goals by sub-goals/other formulas, we generate induction formulas over recursive definitions and add these induction formulas as additional properties to the search space.

When compared to superposition approaches treating certain E-theories [19] or function definitions as rewrite rules [16], we note that our method designs new induction inference rules as simplification rules in superposition and strengthens induction hypotheses during saturation-based inductive reasoning. Our approach extends [17] by handling recursive definitions as rewrite rules and multiple clauses in a single induction step; the latter is often required when assumptions are supported in universally quantified conjectures. Unlike [16], our technique generalizes to scenarios where multiple induction steps are needed to refute non-equality literals. Contrarily to [12], we are not limited to induction over term algebras as most of these techniques work for e.g. mathematical induction as well.

While our approach often does not need auxiliary lemmas due to generalizations over (complex) term occurrences and strengthened induction hypotheses, extending our work towards lemma generation would be beneficial. In particular, theory exploration and lemma generation approaches from [8], [15], [10], [34], [35], [13] could complement our method, ranging from randomly generating terms by iterative deepening to analysing failed induction steps and even circumventing the need for auxiliary lemmas by using predicates.

IX. Conclusion

We introduce a new approach for automating induction with recursive definition in first-order theorem proving. We design new inference rules for rewriting with function definitions as well as induction hypotheses in superposition-based proving. We generate induction formulas based on recursive function definitions and extend our work to support multi-clause induction. Our experiments show that induction with recursive definitions in superposition allows us to solve many new problems that other automated reasoners failed to prove.

Acknowledgments

This work was partially funded by the ERC CoG ARTIST 101002685, the ERC StG SYMCAR 639270, the EPSRC grant EP/P03408X/1, the FWF grant LogiCS W1255-N23, the Amazon ARA 2020 award FOREST and the TU Wien SecInt DK.

REFERENCES

[1] B. Cook, "Formal Reasoning About the Security of Amazon Web Services," in *CAV*, H. Chockler and G. Weissenbacher, Eds. Springer, 2018, pp. 38–47.

[2] "SHA-2 Cryptographic Hash Standard," National Institute of Standards and Technology, 2002. [Online]. Available: https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2withchangenotice.pdf

[3] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *S&P*, 1982, pp. 11–20.

[4] A. Bundy, "The Automation of Proof by Mathematical Induction," in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds., 2001, vol. I, ch. 13, pp. 845–911.

[5] J. S. Moore and C.-P. Wirth, "Automation of Mathematical Induction as part of the History of Logic," *CoRR*, vol. abs/1309.6226, 2013. [Online]. Available: http://arxiv.org/abs/1309.6226

[6] W. Sonnex, S. Drossopoulou, and S. Eisenbach, "Zeno: An automated prover for properties of recursive data structures," in *TACAS*, C. Flanagan and B. König, Eds. Springer, 2012, pp. 407–421.

[7] R. S. Boyer and J. S. Moore, *A Computational Logic Handbook*. Academic Press, 1988.

[8] A. Bundy, D. Basin, D. Hutter, and A. Ireland, *Rippling: Meta-Level Guidance for Mathematical Reasoning*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.

[9] G. O. Passmore, S. Cruanes, D. Ignatovich, D. Aitken, M. Bray, E. Kagan, K. Kanishev, E. Maclean, and N. Mometto, "The Imandra Automated Reasoning System (System Description)," in *IJCAR*, N. Peltier and V. Sofronie-Stokkermans, Eds. Springer, 2020, pp. 464–471.

[10] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating Inductive Proofs Using Theory Exploration," in *CADE*, M. P. Bonacina, Ed. Springer, 06 2013, pp. 392–406.

[11] I. L. Valbuena and M. Johansson, "Conditional Lemma Discovery and Recursion Induction in Hipster," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 72, 2015. [Online]. Available: https://doi.org/10.14279/tuj.eceasst.72.1009

[12] M. Echenheim and N. Peltier, "Combining Induction and Saturation-Based Theorem Proving," *J. Automated Reasoning*, vol. 64, pp. 253–294, 2020.

[13] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, "Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates," in *IJCAR*, N. Peltier and V. Sofronie-Stokkermans, Eds. Springer, 2020, pp. 83–102.

[14] L. Kovács, S. Robillard, and A. Voronkov, "Coming to Terms with Quantified Reasoning," in *POPL*, G. Castagna and A. D. Gordon, Eds., 2017, pp. 260–270.

[15] A. Reynolds and V. Kuncak, "Induction for SMT Solvers," in *VMCAI*, D. D'Souza, A. Lal, and K. G. Larsen, Eds. Springer, 2015, pp. 80–98.

[16] S. Cruanes, "Superposition with Structural Induction," in *FroCoS*, C. Dixon and M. Finger, Eds. Springer, 2017, pp. 172–188.

[17] G. Reger and A. Voronkov, "Induction in saturation-based proof search," in *CADE*, P. Fontaine, Ed. Springer, 2019, pp. 477–494.

[18] P. Hozzová, M. Hajdú, L. Kovács, J. Schoisswohl, and A. Voronkov, "Induction with Generalization in Superposition Reasoning," in *CICM*, C. Benzmüller and B. Miller, Eds. Springer, 2020, pp. 123–137.

[19] R. Nieuwenhuis and A. Rubio, "Paramodulation-Based Theorem Proving," in *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds., 2001, vol. I, ch. 7, pp. 371–443.

[20] S. Schulz, S. Cruanes, and P. Vukmirović, "Faster, Higher, Stronger: E 2.3," in *CADE*, P. Fontaine, Ed. Springer, 2019, pp. 495–507.

[21] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski, "SPASS Version 3.5," in *CADE*, R. A. Schmidt, Ed. Springer, 2009, pp. 140–145.

[22] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *CAV*, N. Sharygina and H. Veith, Eds. Springer, 2013, pp. 1–35.

[23] A. Voronkov, "AVATAR: The Architecture for First-Order Theorem Provers," in *CAV*, A. Biere and R. Bloem, Eds. Springer, 2014, pp. 696–710.

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

[25] L. Dixon and J. Fleuriot, "IsaPlanner: A Prototype Proof Planner in Isabelle," in *CADE*, F. Baader, Ed. Springer, 2003, pp. 279–283.

[26] C. Walther, "Computing Induction Axioms," in *LPAR*, A. Voronkov, Ed. Springer, 1992, pp. 381–392.

[27] D. Beyer, S. Löwe, and P. Wendler, "Reliable Benchmarking: Requirements and Solutions," *Int. J. on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 1–29, 2019.

[28] D. Beyer, "Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016)," in *TACAS*, M. Chechik and J.-F. Raskin, Eds. Springer, 2016, pp. 887–904.

[29] N. Smallbone, M. Johansson, and K. Claessen, "Tons of Inductive Problems (TIP)," `tip-org.github.io`, Springer, pp. 333–337, 2015.

[30] M. Hajdu, P. Hozzová, L. Kovács, J. Schoisswohl, and A. Voronkov, "Inductive Benchmarks for Automated Reasoning," in *CICM*, 2021, to appear. [Online]. Available: https://easychair.org/publications/preprint/gGb9

[31] G. Reger, M. Suda, and A. Voronkov, "New Techniques in Clausal Form Generation," in *GCAI*, C. Benzmüller, G. Sutcliffe, and R. Rojas, Eds., 2016, pp. 11–23.

[32] B. Gleiss and M. Suda, "Layered Clause Selection for Theory Reasoning," in *IJCAR*, N. Peltier and V. Sofronie-Stokkermans, Eds. Springer, 2020, pp. 402–409.

[33] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, G. Gopalakrishnan and S. Qadeer, Eds. Springer, 2011, pp. 171–177.

[34] E. Singher and S. Itzhaky, "Theory Exploration Powered By Deductive Synthesis," *ArXiv*, vol. abs/2009.04826, 2020.

[35] A. Murali, L. Peña, C. Löding, and P. Madhusudan, "Synthesizing Lemmas for Inductive Reasoning," *ArXiv*, vol. abs/2009.10207, 2020.