EPiC Series in Computing
Volume 100, 2024, Pages 278–294
Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Rewriting and Inductive Reasoning

Márton Hajdu, Laura Kovács, and Michael Rawson

TU Wien
{marton.hajdu,laura.kovacs,michael.rawson}@tuwien.ac.at

Abstract

Rewriting techniques based on reduction orderings generate "just enough" consequences to retain first-order completeness. This is ideal for superposition-based first-order theorem proving, but for at least one approach to inductive reasoning we show that we are missing crucial consequences. We therefore extend the superposition calculus with rewriting-based techniques to generate sufficient consequences for automating induction in saturation. When applying our work within the unit-equational fragment, our experiments with the theorem prover Vampire show significant improvements for inductive reasoning.

1 Introduction

Automating *proof by induction* is a particularly hard task with a long history [1, 6, 8, 26, 37]. A promising line of research in this direction comes with the integration of induction into saturation-based first-order theorem proving [12, 16, 20, 33, 38], by extending the logical calculus with induction inference rules. Such induction rules are of the form

$$\frac{\neg L[t] \vee C}{\mathsf{cnf}(\neg F \vee C)}$$

where $\neg L[t] \vee C$ is a clause, $\neg L[t]$ a ground literal, t a term of some inductive data type, and $F \to \forall x.L[x]$ is an instance of some valid second-order induction *schema*, for example structural induction[1]. Note that the schema instance $F \to \forall x.L[x]$ is applied to resolve $\neg L[t]$, but the schema instance is never added to the saturation search space, making sure that the conclusion of the inference is derived via inductive reasoning. As such, the application of the induction rule is *triggered* by the presence of the literal $\neg L[t]$ in the search space. Hence, if there is no such literal, no induction is applied. Applying induction only when triggered means that only premises of schema instances directly related to a clause selected during proof search are added to the search space. This is a strong heuristic method for automating induction, particularly in the presence of full first-order logic and theories [21, 23, 24].

Challenge. Unfortunately, applying induction only when triggered leads to tension between two competing factors:

[1] $\mathsf{cnf}(\neg F \vee C)$ denotes the clausified formula $\neg F \vee C$

(i) It may be that a valid goal is not provable without deriving a certain consequence C, which in turn triggers an induction rule with premise C with a specific schema.

(ii) Efficient first-order calculi, such as superposition [3], go to great lengths to derive and retain only those consequences absolutely required for refutational completeness. Therefore, induction rules that could be triggered by a (missing) consequence C might not be applied.

We show in our motivating example that this tension causes a lost proof in practice (Section 2), as superposition may avoid generating consequences that would be needed to be used in inductive proofs.

Our contributions. For automating (triggered) induction in saturation, we thus need to do something counter-intuitive for those accustomed to first-order superposition reasoning. We propose deriving slightly *more* consequences than usual, in order to trigger induction rules with suitable schemas. Naturally, this must still be as few extra consequences as possible in order to retain a high level of first-order efficiency. Concretely, motivated by applications of program verification [17] in this paper we focus on the unit-equational fragment of first-order logic with induction and bring the following contributions.

(1) We introduce a modified superposition calculus (Section 4) with slightly relaxed constraints, allowing us to derive consequences that cannot be derived by standard superposition.

(2) We impose new restrictions on our calculus (Section 4) to further improve efficiency while retaining our newfound ability to generate consequences.

(3) We improve redundancy elimination for saturation with induction, providing sufficient conditions for skipping redundant induction steps and rewrites (Section 5).

(4) We implement our approach in the Vampire first-order prover [28] and apply it to proving inductive properties (Section 6). Results show that our work improves the state of the art in automating proofs by induction.

2 Motivating Example

We motivate our work from the domain of open programs [29, 32] using Figure 1. The data types nat and list—corresponding to the term algebras of natural numbers and lists in first-order logic—are defined inductively using constructors as given by the first-order formulas (nat.1)–(list.3). Moreover, Figure 1 defines the recursive functions for the addition of natural numbers $(+)$, list append $(@)$ and list length $(\llbracket . \rrbracket)$, encoded by (plus.1)–(length.2), and declares the uninterpreted functions f and g corresponding to two open programs whose behaviour is given by axioms (ax.1)–(ax.3). We use infix notation for the symbols $+$, $@$ and $\llbracket . \rrbracket$. All properties in Figure 1 are implicitly universally-quantified.

Suppose we are trying to prove that the axioms (denoted Ax) in Figure 1 imply the following first-order formula:

$$\forall x, y. \llbracket g(y) \mathbin{@} f(x) \rrbracket \simeq \mathsf{s}(\llbracket x \rrbracket + \llbracket y \rrbracket) \tag{Conj}$$

Proving (Conj) in classical first-order logic can be reduced to establishing the unsatisfiability of the negation of (Conj) together with the axioms of Figure 1. That is, we prove unsatisfiability of (nConj) together with the axioms (nat.1)–(ax.3); here, (nConj) is the negated and Skolemized form, of (Conj), using the Skolem (list) functions c, d. While the axioms imply formula (Conj) in the theory of lists and natural numbers, $\mathsf{Ax} \wedge$(nConj) is not first-order unsatisfiable. Showing

Signature

$\mathsf{0} : \mathsf{nat}$	$\mathsf{cons} : \mathsf{nat} \to \mathsf{list} \to \mathsf{list}$	$(\llbracket . \rrbracket) : \mathsf{list} \to \mathsf{nat}$
$\mathsf{s} : \mathsf{nat} \to \mathsf{nat}$	$(+) : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$	$f : \mathsf{list} \to \mathsf{list}$
$\mathsf{nil} : \mathsf{list}$	$(@) : \mathsf{list} \to \mathsf{list} \to \mathsf{list}$	$g : \mathsf{list} \to \mathsf{list}$

Axioms — Ax

$\mathsf{0} \not\simeq \mathsf{s}(x)$	(nat.1)	$\mathsf{nil} \mathbin{@} x \simeq x$	(append.1)
$\mathsf{s}(x) \not\simeq \mathsf{s}(y) \lor x \simeq y$	(nat.2)	$\mathsf{cons}(x, y) \mathbin{@} z \simeq \mathsf{cons}(x, y \mathbin{@} z)$	(append.2)
$\mathsf{nil} \not\simeq \mathsf{cons}(x, y)$	(list.1)	$\llbracket \mathsf{nil} \rrbracket \simeq \mathsf{0}$	(length.1)
$\mathsf{cons}(x, y) \not\simeq \mathsf{cons}(z, u) \lor x \simeq z$	(list.2)	$\llbracket \mathsf{cons}(x, y) \rrbracket \simeq \mathsf{s}(\llbracket y \rrbracket)$	(length.2)
$\mathsf{cons}(x, y) \not\simeq \mathsf{cons}(z, u) \lor y \simeq u$	(list.3)	$g(x) \mathbin{@} f(y) \simeq f(x) \mathbin{@} g(y)$	(ax.1)
$\mathsf{0} + x \simeq x$	(plus.1)	$\llbracket x \rrbracket + \llbracket f(y) \rrbracket \simeq \llbracket f(x) \rrbracket + \llbracket y \rrbracket$	(ax.2)
$\mathsf{s}(x) + y \simeq \mathsf{s}(x + y)$	(plus.2)	$\llbracket f(g(x)) \rrbracket \simeq \mathsf{s}(\llbracket x \rrbracket)$	(ax.3)

Negated conjecture — $\neg\mathsf{Conj}$		**Auxiliary lemma** — Lem	
$\llbracket g(d) \mathbin{@} f(c) \rrbracket \not\simeq \mathsf{s}(\llbracket c \rrbracket + \llbracket d \rrbracket)$	(nConj)	$\forall x, y. \llbracket x \mathbin{@} y \rrbracket \simeq \llbracket y \rrbracket + \llbracket x \rrbracket$	(lemma)

Figure 1: Motivating example, conjecturing that the first-order formula Conj is implied by first-order axioms Ax.

unsatisfiability requires an additional first-order axiom over lists, in particular the *auxiliary lemma* Lem of Figure 1. With this lemma, $\mathsf{Ax} \land (\mathsf{nConj}) \land \mathsf{Lem}$ is unsatisfiable and validity of Conj follows.

Let us make two key observations. First, from (nConj), the negation of an instance of Lem (denoted by NLem) can be derived via rewriting with equal terms. Second, Lem is not a first-order consequence of Ax, but it is valid with respect to Ax in the background theory of lists and natural numbers, a fact that can be shown by induction. We make use of these two observations to synthesize and use Lem in the proof of Conj as follows:

(i) We derive NLem from $\mathsf{Ax} \land (\mathsf{nConj})$ by rewriting. Soundness of rewriting ensures that the unsatisfiability of $\mathsf{Ax} \land \mathsf{NLem}$ implies the unsatisfiability of $\mathsf{Ax} \land (\mathsf{nConj})$.

(ii) We refute $\mathsf{Ax} \land \mathsf{NLem}$ by instantiating a valid induction schema with Lem to obtain a valid first-order induction *axiom*, which in conjunction with $\mathsf{Ax} \land \mathsf{NLem}$ is unsatisfiable, implying the unsatisfiability of $\mathsf{Ax} \land (\mathsf{nConj})$ and hence the claim of Figure 1.

To derive NLem from $\mathsf{Ax} \land (\mathsf{nConj})$, we apply the following rewriting steps:

– rewrite (nConj)	with (plus.2) resulting in	$\llbracket g(d) \mathbin{@} f(c) \rrbracket \not\simeq \mathsf{s}(\llbracket c \rrbracket) + \llbracket d \rrbracket$	(1)
– rewrite (1)	with (ax.3) resulting in	$\llbracket g(d) \mathbin{@} f(c) \rrbracket \not\simeq \llbracket f(g(c)) \rrbracket + \llbracket d \rrbracket$	(2)
– rewrite (2)	with (ax.2) resulting in	$\llbracket g(d) \mathbin{@} f(c) \rrbracket \not\simeq \llbracket g(c) \rrbracket + \llbracket f(d) \rrbracket$	(3)
– rewrite (3)	with (ax.1) resulting in	$\llbracket f(d) \mathbin{@} g(c) \rrbracket \not\simeq \llbracket g(c) \rrbracket + \llbracket f(d) \rrbracket$	(4)

Notice that clause (4) is the negation of Lem, instantiated with $f(d)$ and $g(c)$. To refute $\mathsf{Ax} \land \mathsf{NLem}$, we *conjecture* Lem to be proven by induction, by taking the negation of (4) and *by generalizing over* the term $f(d)$. Hence, we instantiate the following *second-order structural induction formula* over lists of natural numbers with Lem:

$$\forall F. \Big(\big(F(\mathsf{nil}) \land \forall x \in \mathsf{nat}, y \in \mathsf{list}. (F(y) \to F(\mathsf{cons}(x, y))) \big) \to \forall z \in \mathsf{list}. F(z) \Big) \tag{5}$$

Showing the first-order unsatisfiability of this induction axiom in conjunction with $\mathsf{Ax} \wedge \mathsf{NLem}$ requires no further rewriting [2].

Note that the above reasoning actually proves Lem *in addition* to Conj, but it comes with the following two main challenges for proving Figure 1:

(C1) use rewriting with equalities to derive NLem from $\mathsf{Ax} \wedge$ (nConj) (Section 4);

(C2) combine inductive reasoning with first-order reasoning to refute $\mathsf{Ax} \wedge \mathsf{NLem}$ (Section 5).

For tackling challenge **(C1)**, we use *rewriting* inferences to rewrite equal terms and generate auxiliary lemmas. However, such proof steps cannot always be performed with the ubiquitous (ordered) superposition inferences. Let us use a Knuth-Bendix simplification ordering (KBO) $\succ$ [30] parameterized by a constant weight function and the precedence $\gg$:

$$d \gg c \gg g \gg f \gg (\llbracket . \rrbracket) \gg (@) \gg (+) \gg \mathsf{cons} \gg \mathsf{nil} \gg \mathsf{s} \gg \mathsf{0}$$

The ordering $\succ$ cannot orient the equalities of ($\mathsf{plus.2}$) and ($\mathsf{ax.3}$) right-to-left so that clause (2) could be derived by rewriting. Moreover, axioms, such as ($\mathsf{ax.1}$)–($\mathsf{ax.3}$) that implicitly specify program functionalities, cannot be oriented to avoid a search space explosion due to superposition inferences. Addressing such obstacles, we introduce an extension of the superposition calculus (Section 4) to enable generating auxiliary lemmas during saturation. Our extension solves challenge **(C1)** and provides an efficient reasoning backend for challenge **(C2)**.

3 Theoretical Background

We assume familiarity with many-sorted first-order logic with equality. Variables are denoted with x, y, z, terms with s, t, u, w, l, r, all possibly with indices. A term is *ground* if it contains no variables. We use the standard logical connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\leftrightarrow$, and quantifiers $\forall$ and $\exists$. A *literal* is an atom or its negation. The literal $\overline{L}$ denotes the complement of literal L. A disjunction of literals is a *clause*. We denote clauses by C, D and reserve the symbol $\square$ for the *empty clause* that is logically equivalent to $\bot$. We refer to the *clausal normal form* of a formula F by $\mathsf{cnf}(F)$. We assume that cnf preserves satisfiability, i.e. F is satisfiable iff $\mathsf{cnf}(F)$ is satisfiable. We use $\simeq$ to denote equality and write $\bowtie$ for either $\simeq$ or $\not\simeq$.

A *position* is a finite sequence of positive integers. The *root position* is the empty sequence, denoted by ϵ. Let p and q be positions. The *concatenation* of p and q is denoted by pq. We say that p is *above* q if there exists a position r such that $pr = q$, denoted by $p \leqslant q$. We say that p and q are *parallel*, denoted by $p \parallel q$, if $p \nleqslant q$ and $q \nleqslant p$. We say that p is *to the left* of q, denoted by $p <_l q$, if there are positive integers i and j, positions r, p' and q' such that $p = rip'$, $q = rjq'$ and $i < j$.

An *expression* E is a term, literal, clause or formula. We write $E[s]_p$ to state that the expression E contains some distinguished occurrence of the term s at some position p. We might simply write $E[s]$ if the position p is not relevant. Further, $E[s \mapsto t]$ denotes that this occurrence of s is replaced with t; when s is clear from the context, we simply write $E[t]$. We say that t is a *subterm* of $s[t]$, denoted by $t \trianglelefteq s[t]$; and a *strict subterm* if additionally $t \neq s[t]$, denoted by $t \triangleleft s[t]$. A *substitution* is a mapping from variables to terms. We denote substitutions by θ, σ, ρ, μ, η. A substitution θ is a *unifier* of two terms s and t if $s\theta = t\theta$, and is a *most general unifier* (denoted $\mathsf{mgu}(s,t)$) if for every unifier η of s and t, there exists a substitution μ s.t. $\eta = \theta\mu$.

[2]See Appendix A of the extended version [22] of this paper for details.

$$
\begin{array}{llll}
(\mathsf{Sup}) & \dfrac{\underline{s[u]} \bowtie t \vee C \quad \underline{l \simeq r} \vee D}{(s[r] \bowtie t \vee C \vee D)\theta} & \text{where} & \begin{array}{l}(1)\ u \text{ is not a variable,}\\ (2)\ \theta = \mathsf{mgu}(l,u),\\ (3)\ r\theta \not\succeq l\theta \text{ and } t\theta \not\succeq s[u]\theta,\end{array}\\[2ex]
(\mathsf{EqRes}) & \dfrac{\underline{s \not\simeq t} \vee C}{C\theta} & \text{where} & \theta = \mathsf{mgu}(s,t),\\[2ex]
(\mathsf{EqFac}) & \dfrac{\underline{s \simeq t} \vee \underline{u \simeq w} \vee C}{(s \simeq t \vee t \not\simeq w \vee C)\theta} & \text{where} & \begin{array}{l}(1)\ \theta = \mathsf{mgu}(s,u),\\ (2)\ t\theta \not\succeq s\theta \text{ and } w\theta \not\succeq t\theta.\end{array}
\end{array}
$$

Figure 2: The superposition calculus **Sup** for first-order logic with equality.

Let $\rightarrow$ be a binary relation. The *inverse* of $\rightarrow$ is denoted by $\leftarrow$. The *reflexive-transitive closure* of $\rightarrow$ is denoted by $\rightarrow^*$. A binary relation $\rightarrow$ over the set of terms is a *rewrite relation* if (i) $l \rightarrow r \Rightarrow l\theta \rightarrow r\theta$ and (ii) $l \rightarrow r \Rightarrow s[l] \rightarrow s[r]$ for any term l, r, s and substitution θ. A *rewrite ordering* is a strict rewrite relation. A *reduction ordering* is a well-founded rewrite ordering. In this paper we consider reduction orderings total on ground terms. Such orderings satisfy $s \triangleright t \Rightarrow s \succ t$ and are also called *simplification orderings*.

3.1 Saturation-Based Theorem Proving

We briefly introduce saturation-based proof search in first-order theorem proving. For details, we refer to [20, 28]. The majority of first-order theorem provers work with clauses, rather than arbitrary formulas. Let $S = \mathcal{A} \cup \{\neg G\}$ be a set of clauses including assumptions $\mathcal{A}$ and the clausified negation $\neg G$ of a goal G. Given S, first-order provers *saturate* S by computing all logical consequences of S with respect to a sound inference system $\mathcal{I}$. This process is called *saturation*. An inference system $\mathcal{I}$ is a set of inference rules of the form

$$\frac{C_1 \quad \dots \quad C_n}{C},$$

where $C_1, \dots, C_n$ are the *premises* and C is the *conclusion* of the inference. We also write $C_1, \dots, C_n \vdash_{\mathcal{I}} C$ to denote an inference in $\mathcal{I}$; as $\mathcal{I}$ is sound, this also means that C is a logical consequence of $C_1, \dots, C_n$. We denote that $\mathcal{I}$ derives clause D from clauses $\mathcal{C}$ with $\mathcal{C} \vdash^*_{\mathcal{I}} D$. If the the saturated set of S contains the empty clause $\square$, the original set S of clauses is unsatisfiable, implying validity of $\mathcal{A} \rightarrow G$; in this case, we establish a *refutation* of $\neg G$ from $\mathcal{A}$.

Completeness and efficiency of saturation-based reasoning relies on selecting/adding clauses from/to S using the inference system $\mathcal{I}$. To constrain the inference system, some first-order provers use simplification orderings on terms. Simplification orderings are extended to orderings over literals and clauses using the bag extension of the ordering; for simplicity, we write $\succ$ both for the term ordering and its clause ordering extensions. 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' implies C and is smaller than $\{C\}$, i.e. $S' \models C$ and $\{C\} \succ S'$.

The *superposition calculus*, denoted **Sup** and given in Figure 2, is the most common inference system used by saturation-based first-order theorem provers [30]. We assume a literal selection function satisfying the standard condition on $\succ$ and underline selected literals in **Sup** inferences. The **Sup** calculus is *sound* and *refutationally complete*: for any unsatisfiable formula $\neg G$, the empty clause $\square$ can be derived as a logical consequence of $\neg G$.

3.2 Inductive Reasoning in Saturation

Inductive reasoning has recently been embedded in saturation-based theorem proving [12,33], by extending **Sup** with a new inference rule. More precisely, we introduce a family of induction inference rules parameterized by a second-order formula G with exactly one free second-order variable F: the formula over which induction should be applied. Moreover, we restrict inductions to a set of terms $\mathcal{I}nd(\mathcal{T}) \subseteq \mathcal{T}$ where $\mathcal{T}$ is the set of all terms. Then, the inference rules are of the following form:

$$(\mathsf{Ind}_G)\ \frac{\overline{L}[t] \vee C}{\mathsf{cnf}(\neg G[L[x]] \vee C)} \quad \text{where} \quad \begin{array}{l}(1)\ L[t] \text{ is ground and } t \in \mathcal{I}nd(\mathcal{T}),\\ (2)\ \forall F.(G[F] \to \forall x.F[x]) \text{ is a valid}\\ \quad\ \text{second-order } \textit{induction schema}.\end{array}$$

By an *induction axiom* we refer to an instance of a valid induction schema. When performing an Ind_G inference, the induction schema $\forall F.(G[F] \to \forall x.F[x])$ is said to be *applied on* the clause $\overline{L}[t] \vee C$, or alternatively speaking $\overline{L}[t] \vee C$ is *inducted upon*; in addition, we also say that we *induct on term t* in clause $\overline{L}[t] \vee C$ with induction schema $\forall F.(G[F] \to \forall x.F[x])$. For example, using the schema (5), we parameterize the Ind_G schema with $G := F[\mathsf{nil}] \wedge \forall x, y.(F[y] \to F[\mathsf{cons}(x, y)])$ and obtain the Ind_G instance:

$$(\mathsf{Ind}_G)\ \frac{\overline{L}[t] \vee C}{\begin{array}{c}\overline{L}[\mathsf{nil}] \vee L[c_y] \vee C\\ \overline{L}[\mathsf{nil}] \vee \overline{L}[\mathsf{cons}(c_x, c_y)] \vee C\end{array}} \quad \text{where} \quad \begin{array}{l}(1)\ L[t] \text{ is ground},\\ (2)\ t \in \mathcal{I}nd(\mathcal{T}) \text{ is of sort } \mathsf{list},\\ (3)\ c_x \text{ and } c_y \text{ are fresh Skolem symbols.}\end{array}$$

Note that the above Ind_G inference instance yields two clauses, where each clause results from the clausification of schema (5) being resolved with the premise.

4 Efficient Rewriting in Saturation

As motivated in Section 2, rewriting derives clauses useful for auxiliary lemma generation that **Sup** is not able to derive. We therefore focus on rewriting variants captured by the following inference rule:

$$(\mathsf{Rw})\ \frac{C[l\theta] \quad l \simeq r}{C[r\theta]}$$

where θ is a substitution. We call an Rw inference a *downward rewrite* if $l\theta \succ r\theta$, and call an Rw inference an *upward rewrite* if $l\theta \prec r\theta$.

We start by defining our base inference system, called the ***Re**writing **C**alculus* (**ReC**), as the calculus extending **Sup** with Rw. In other words, we define **ReC** to consists of the inference rules of **Sup**$\cup\{\mathsf{Rw}\}$. The refutational completeness of **ReC** follows from the completeness of its subsystem **Sup**. In addition to completeness, we consider the following property over inference systems, and in particular over **ReC**.

Definition 1 (Equational derivability (ED))**.** Let θ be a substitution. An inference system $\mathcal{I}$ *admits equational derivability (ED)* if, for any set of equations $\mathcal{C}$, equation $l \simeq r$ and clause $D[l\theta]$, $\mathcal{C} \vdash^*_{\mathcal{I}} D[l\theta]$ implies $\mathcal{C}, l \simeq r \vdash^*_{\mathcal{I}} D[r\theta]$.

Equational derivability in Definition 1 essentially expresses that an inference system can simulate the application of the Rw rule, by some (possibly longer) derivation. This allows us to introduce and compare variants of **ReC**, by imposing additional rewriting constraints in Rw. We state the following, straightforward result.

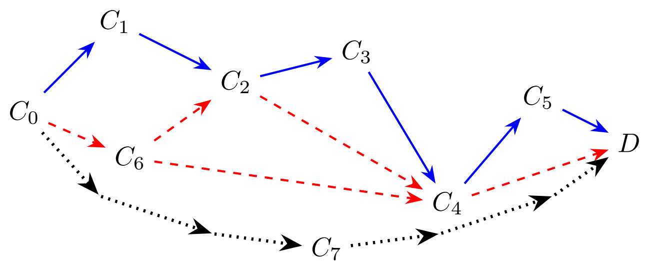

Figure 3: Possible rewrite sequences from a ground clause C_0 to a ground clause D. The total order between clauses is $C_1 \succ C_3 \succ C_2 \succ C_5 \succ C_0 \succ D \succ C_6 \succ C_4 \succ C_7$, as also visualized by the vertical alignment of clauses.

Theorem 1 (**ReC**–ED)**.** *The inference system* **ReC** *admits ED.*

In the sequel, we develop three improved variants of **ReC** that admit ED, and thus derive the same consequences with equations as **ReC** does.

4.1 Peak Elimination in ReC

Let $\mathcal{C}$ be a satisfiable set of clauses. Suppose there is some ground clause D that triggers the generation of a necessary inductive axiom, and suppose D can be derived from $\mathcal{C}$ via rewrites with equations in $\mathcal{C}$. Hence our goal is to derive D. In Figure 3, we show[3] possible ways to derive D from a ground clause $C_0 \in \mathcal{C}$ using equations in $\mathcal{C}$. Arrows of Figure 3 point in the direction of deduction. Assume that all clauses in Figure 3 are ground; using a total simplification ordering $\succ$ over ground clauses, we order clauses in Figure 3 as given by their vertical alignment in Figure 3. Therefore, an arrow going vertically upwards (resp. downwards) in Figure 3 corresponds to an upward (resp. downward) rewrite variant of Rw. We use three different arrows in Figure 3, corresponding to paths available at different saturation steps (iterations) while saturating $\mathcal{C}$:

(1) Arrows $\longrightarrow$ designate a path which is possible in a certain iteration i during saturation, that is with equations available at iteration i.

(2) Arrows $\dashrightarrow$ correspond to paths in later iterations than i but not necessarily at the end of the saturation process.

(3) Arrows $\cdots\!\!\blacktriangleright$ correspond to the "ideal path" at the end of the saturation process, that is, when the equations are transformed into a set of equations corresponding to a complete (confluent and terminating) rewrite system.

As shown by the many rewriting steps of Figure 3, choosing a path between C_0 and D is not trivial. For example, using arrows $\longrightarrow$, we may derive D from C_0 in iteration i already, but in principle we have to exhaustively apply rewrites in all "directions", resulting in many duplicate clauses. A different strategy is to wait until saturation end, in which case using arrows $\cdots\!\!\blacktriangleright$ we rewrite C_0 into its normal form C_7, and then from C_7 we reach D only by upward rewrites. However, saturation may never terminate, for example in the presence of associativity and commutativity (AC) axioms.

[3]similarly to [35].

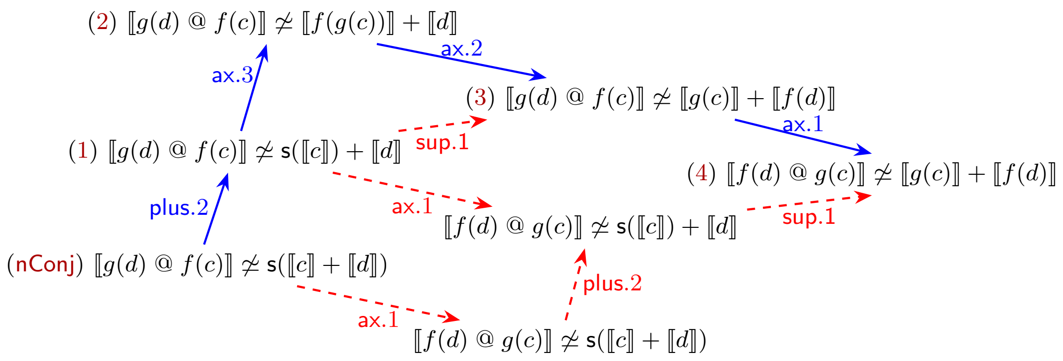
(2) $[\![g(d) \mathbin{@} f(c)]\!] \not\simeq [\![f(g(c))]\!] + [\![d]\!]$
ax.3
ax.2
(3) $[\![g(d) \mathbin{@} f(c)]\!] \not\simeq [\![g(c)]\!] + [\![f(d)]\!]$
ax.1
(1) $[\![g(d) \mathbin{@} f(c)]\!] \not\simeq \mathsf{s}([\![c]\!]) + [\![d]\!]$
sup.1
(4) $[\![f(d) \mathbin{@} g(c)]\!] \not\simeq [\![g(c)]\!] + [\![f(d)]\!]$
plus.2
ax.1
$[\![f(d) \mathbin{@} g(c)]\!] \not\simeq \mathsf{s}([\![c]\!]) + [\![d]\!]$
sup.1
(nConj) $[\![g(d) \mathbin{@} f(c)]\!] \not\simeq \mathsf{s}([\![c]\!] + [\![d]\!])$
plus.2
ax.1
$[\![f(d) \mathbin{@} g(c)]\!] \not\simeq \mathsf{s}([\![c]\!] + [\![d]\!])$

Figure 4: Possible rewrite sequences to derive clause (4) from (nConj) in the example of Section 2.

Another option is to find a path of a specific form during saturation, such as the paths designated by $\dashrightarrow$ arrows in Figure 3. We propose to avoid so-called *peaks* during saturation, where a peak comes with an upward rewrite followed by a downward rewrite. That is, upward rewrites followed by downward rewrites should be avoided. Depending on the positions in which the upward and downward rewrites happen, the following two possibilities occur:

(i) If the positions of upward and downward rewrites are *parallel*, the two rewrites can be simply flipped. For example, the path $C_0 \longrightarrow C_1 \longrightarrow C_2$ of Figure 3 is replaced by the path $C_0 \dashrightarrow C_6 \dashrightarrow C_2$.

(ii) If the positions are *overlapping*, there is a superposition between the two rewriting equations of the peak. This superposition inference generates an equation that "cuts" the peak, giving a one-step rewrite alternative instead of two rewrites. For example, the peak $C_4 \longrightarrow C_5 \longrightarrow D$ can be replaced by $C_4 \dashrightarrow D$. Note that sometimes multiple superpositions have to be performed before the path can be continued, e.g. the (double) peak $C_6 \dashrightarrow C_2 \longrightarrow C_3 \longrightarrow C_4$ needs two superpositions to be eliminated, and performed simply as $C_6 \dashrightarrow C_4$.

To avoid peaks in saturation, we distinguish clauses resulting from upward rewrites (annotated as $^{\blacksquare}C$) from other clauses (annotated as $^{\square}C$). We use the notation $^{◪}C$ to denote either of these. We might leave clauses without annotation if this information is not relevant in the context. We split the Rw inference into two components, resulting in the following inferences:

$$(\mathsf{Rw}_{\downarrow})\ \frac{^{\square}C[l\theta] \quad ^{\square}l \simeq r}{^{\square}C[r\theta]} \quad \text{where } l\theta \not\preceq r\theta, \qquad (\mathsf{Rw}_{\uparrow})\ \frac{^{◪}C[l\theta] \quad ^{\square}l \simeq r}{^{\blacksquare}C[r\theta]} \quad \text{where } l\theta \not\succeq r\theta.$$

We denote our **ReC** *variant for avoiding peaks in saturation* by $\mathbf{ReC}_{\vee}$, and define $\mathbf{ReC}_{\vee}$ to consist of the inference rules of $\mathbf{Sup} \cup \{\mathsf{Rw}_{\downarrow}, \mathsf{Rw}_{\uparrow}\}$.

Remark 1. *Note that the* $\mathsf{Rw}_{\downarrow}$ *and* $\mathsf{Rw}_{\uparrow}$ *rules both allow rewriting with incomparable equations. The reason for this is that disallowing rewrites with incomparable equations after upward rewrites violates ED in some cases* [4].

[4]See Appendix B of our extended paper [22] for details.

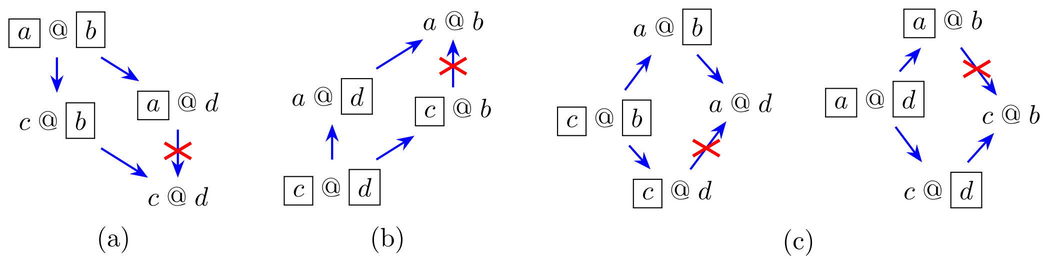

Figure 5: Possible parallel rewrites in **ReC** given equations $a \simeq c$, $b \simeq d$ with $a \succ c$ and $b \succ d$. Rewrites corresponding to crossed out arrows are not performed in the left-to-right order.

Example 1. *Solid blue lines ($\longrightarrow$) in Figure 4.1 show the sequence of rewrite steps to reach clause (4) from clause (nConj) within the motivating example of Section 2, when using* **ReC***.*

Alternatively, we can perform the following $\dashrightarrow$ steps with $\mathbf{ReC}_\vee$*. A superposition into clause (ax.2) with (ax.3) results in*

$$[\![g(x)]\!] + [\![f(y)]\!] \simeq \mathsf{s}([\![x]\!]) + [\![y]\!]. \qquad \text{(sup.1)}$$

Using clause (sup.1), we eliminate the peak through clause (2), and directly derive clause (3) from clause (1) in $\mathbf{ReC}_\vee$*. Note that we can switch the order of rewrites using clauses (ax.1) and (plus.2); and similarly the order of rewrites using clauses (ax.1) and (sup.1). We thus obtain the* $\mathbf{ReC}_\vee$ *derivation of clause (4) via rewriting clause (nConj) with (ax.1), then with (plus.2), and finally with (sup.1).* □

We conclude our presentation of $\mathbf{ReC}_\vee$ with the following result.

Theorem 2 ($\mathbf{ReC}_\vee$–ED)**.** *The inference system* $\mathbf{ReC}_\vee$ *admits ED.*

4.2 Diamond Elimination in ReC

Note that rewrites in parallel positions can be performed in any order. If the rewriting is performed in all possible orders, this leads to a large number of duplicated clauses. In this section, we restrict **ReC** and $\mathbf{ReC}_\vee$ to eliminate this effect, while preserving equational derivability from Definition 1.

Figure 5 illustrates rewriting possibilities for two parallel rewrites with **ReC**. Depending on the direction of the rewrites, there are three possibilities, that is, three "diamonds": Figure 5(a) shows two downward rewrites; Figure 5(b) illustrates two upward rewrites; while Figure 5(c) shows one downward and one upward rewrite. Note that Figure 5(c) contains two subcases, one where the upward rewrite happens in the left position and one where it happens in the right position. We denote the positions to be rewritten with boxes, e.g. $\boxed{a}$. To generate all terms in these diamonds without duplicating any terms, we follow the tradition of reduction strategies in programming languages, for example *leftmost-outermost* (also *call-by-need*) and *leftmost-innermost* (also *call-by-value*) strategies [2]. We choose a *left-to-right* rewriting order, that is, we cannot perform a rewrite to the left of the previous rewrite. In Figure 5, the skipped rewrites are crossed out in red.

Figure 5(a) and Figure 5(b) are the same in $\mathbf{ReC}_\vee$. However, $\mathbf{ReC}_\vee$ avoids duplication of Figure 5(c) in the first place (recall the parallel positions in Figure 3). Therefore, in the case of $\mathbf{ReC}_{\vee}$, we only apply the left-to-right order in the case of multiple consecutive $\mathsf{Rw}_{\downarrow}$ (resp. $\mathsf{Rw}_{\uparrow}$) inferences. Towards this, we associate with each clause C a position p where the previous rewrite was performed. We denote such clauses by ${}_{p}C$. Our modified Rw rule for avoiding duplicated diamonds is:

$$(\mathsf{Rw}^{\rightarrow})\ \frac{{}_{p}C[l\theta]_q \quad l \simeq r}{{}_{q}C[r\theta]_q} \qquad \text{where } q \not<_l p.$$

Our $\mathbf{ReC}$ variant for *avoiding duplicated diamonds during rewritings* is denoted by $\mathbf{ReC}^{\rightarrow}$ and is defined to be $\mathbf{Sup} \cup \{\mathsf{Rw}^{\rightarrow}\}$. We state the following result.

Theorem 3 ($\mathbf{ReC}^{\rightarrow}$–ED)**.** *The inference system* $\mathbf{ReC}^{\rightarrow}$ *admits ED.*

Finally, we define a $\mathbf{ReC}$ variant that *combines peak-elimination with left-to-right rewriting orders.* We denote this $\mathbf{ReC}$ variant by $\mathbf{ReC}^{\rightarrow}_{\vee}$. Here, we enforce the left-to-right order separately between downward and upward rewrites, as captured via the following Rw variants:

$$(\mathsf{Rw}^{\rightarrow}_{\downarrow})\ \frac{{}^{\square}_{p}C[l\theta]_q \quad {}^{\square}l \simeq r}{{}^{\square}_{q}C[r\theta]_q} \qquad \text{where} \quad \begin{array}{l}(1)\ l\theta \not\preceq r\theta,\\ (2)\ q \not<_l p,\end{array}$$

$$(\mathsf{Rw}^{\rightarrow}_{\uparrow})\ \frac{{}^{◪}_{p}C[l\theta]_q \quad {}^{\square}l \simeq r}{{}^{\blacksquare}_{q}C[r\theta]_q} \qquad \text{where} \quad \begin{array}{l}(1)\ l\theta \not\succeq r\theta,\\ (2)\ {}^{◪}_{p}C[l\theta] = {}^{\square}_{p}C[l\theta] \text{ or } q \not<_l p.\end{array}$$

Our inference system $\mathbf{ReC}^{\rightarrow}_{\vee}$ is defined as $\mathbf{Sup} \cup \{\mathsf{Rw}^{\rightarrow}_{\downarrow}, \mathsf{Rw}^{\rightarrow}_{\uparrow}\}$ and has the following property.

Theorem 4 ($\mathbf{ReC}^{\rightarrow}_{\vee}$–ED)**.** *The inference system* $\mathbf{ReC}^{\rightarrow}_{\vee}$ *admits ED.*

5 Redundancy and Induction

As mentioned in Section 1, the **Sup** calculus tries to derive and retain as few clauses as possible without losing (refutational) completeness. Within the **ReC** calculus, as well as within its three refinements $\mathbf{ReC}_{\vee}$, $\mathbf{ReC}^{\rightarrow}$ and $\mathbf{ReC}^{\rightarrow}_{\vee}$, however, we not only derive more consequences, but we also avoid simplifications, resulting in less efficient reasoning than via **Sup**. The situation gets even worse when the prolific Ind_G rule is used to enable inductive reasoning.

As a remedy, this section integrates redundancy elimination within our **ReC** calculi extended with Ind_G inferences. Our main goal is to be as efficient as possible without losing inductive proofs. This includes, for example, preserving the ED property for our calculi extended with Ind_G. We introduce sufficient criteria to skip induction inferences in **ReC**, and weaken the ED restriction to avoid deriving useless clauses for first-order and inductive reasoning.

We identify induction inferences that can be omitted without losing proofs and provide efficient ways to detect such (redundant) inferences.

Remark 2. *Note that constructor-based induction schemas give rise to a few optimisations:*

(1) Inducting on t in $L[t] \vee C$ where t has zero occurrences is possible, but using constructor-based induction schemas only results in tautological clauses $L \vee \neg L \vee C$ and clauses with duplicate literals $L \vee L \vee C$. We thus omit inducting on t in $L[t] \vee C$ where t has zero occurrences.

(2) Inducting on base constructors, such as $\mathsf{0}$ *and* nil*, only give weaker forms of the same clauses. For example, inducting upon* nil *in* $L[\mathsf{nil}] \vee C$ *yields clauses of the form* $L[\mathsf{nil}] \vee C' \vee C$*. We thus omit induction on base costructors.*

While the inductive inferences described in Remark 2 can easily be detected, this is not the case with more complex but useless inductive inferences, as shown in the next example.

Example 2. *Consider the* Ind_G *inferences on term* c *in clauses* (nConj) *and* (2)*, respectively. The first* Ind_G *inference yields clauses*

$$[\![g(d) \mathbin{@} f(\mathsf{nil})]\!] \not\simeq \mathsf{s}([\![\mathsf{nil}]\!] + [\![d]\!]) \vee [\![g(d) \mathbin{@} f(c_2)]\!] \simeq \mathsf{s}([\![c_2]\!] + [\![d]\!])$$
$$[\![g(d) \mathbin{@} f(\mathsf{nil})]\!] \not\simeq \mathsf{s}([\![\mathsf{nil}]\!] + [\![d]\!]) \vee [\![g(d) \mathbin{@} f(\mathsf{cons}(c_1, c_2))]\!] \not\simeq \mathsf{s}([\![\mathsf{cons}(c_1, c_2)]\!] + [\![d]\!])$$

where c_1 *and* c_2 *are fresh Skolem constants. The second* Ind_G *inference yields clauses*

$$[\![g(d) \mathbin{@} f(\mathsf{nil})]\!] \not\simeq [\![f(g(\mathsf{nil}))]\!] + [\![d]\!] \vee [\![g(d) \mathbin{@} f(c_4)]\!] \simeq [\![f(g(c_4))]\!] + [\![d]\!]$$
$$[\![g(d) \mathbin{@} f(\mathsf{nil})]\!] \not\simeq [\![f(g(\mathsf{nil}))]\!] + [\![d]\!] \vee [\![g(d) \mathbin{@} f(\mathsf{cons}(c_3, c_4))]\!] \not\simeq [\![f(g(\mathsf{cons}(c_3, c_4)))]\!] + [\![d]\!]$$

where c_3 *and* c_4 *are fresh Skolem constants. The induction formulas used in the two* Ind_G *inferences are equivalent w.r.t. axioms* (nat.1)*–*(ax.3)*, hence it is sufficient to perform only one inference and retain the corresponding conclusions. This is unfortunately hard to detect due to the different sets of Skolem constants* c_1, c_2 *and* c_3, c_4*. For example, simplifying the induction formulas and checking them for equivalence before clausification takes considerable effort.* □

For detecting redundancies similar to Example 2, we characterize redundant induction inferences of interest and introduce sufficient conditions to efficiently detect them.

Definition 2 (Redundant Ind_G inference)**.** Let $\mathcal{I}$ be an inference system that admits ED, C a clause and F a formula. The Ind_G inference $C \vdash \mathsf{cnf}(F)$ is *redundant* in $\mathcal{I} \cup \{\mathsf{Ind}_G\}$ w.r.t. a set of equations E and a clause C', if $C \succ C'$ and there is a formula F' and an Ind_G inference $C' \vdash \mathsf{cnf}(F')$ s.t. F and F' are equivalent modulo rewriting with E.

It is easy to see that only non-redundant Ind_G inferences need to be performed to retain equational consequences and first-order refutations. The following two lemmas show sufficient conditions to efficiently check for redundant induction inferences.

Lemma 1 (Redundant Ind_G – Condition I)**.** *Let* $l \simeq r$ *and* $\overline{L}[t] \vee C$ *be clauses,* x *a fresh variable, and* $\mathcal{I}$ *an inference system that admits ED. If there is a substitution* θ *s.t.* $l\theta \lhd L[x]$ *and* $l\sigma \succ r\sigma$ *where* $\sigma = \theta \cdot \{x \mapsto t\}$*, then the* Ind_G *inference*

$$\overline{L}[t] \vee C \vdash \mathsf{cnf}(\neg G[L[x]] \vee C)$$

is redundant in $\mathcal{I} \cup \{\mathsf{Ind}_G\}$ *w.r.t. the clauses* $l \simeq r$ *and* $(\overline{L}[t])[l\sigma \mapsto r\sigma] \vee C$*.*

Lemma 2 (Redundant Ind_G – Condition II)**.** *Let* $l \simeq r$ *and* $\overline{L}[t] \vee C$ *be clauses,* x *a fresh variable, and* $\mathcal{I}$ *an inference system that admits ED. If there is a substitution* θ *s.t.* $l\theta \unlhd t$ *and* $l\theta \succ r\theta$*, then the* Ind_G *inference*

$$\overline{L}[t] \vee C \vdash \mathsf{cnf}(\neg G[L[x]] \vee C)$$

is redundant in $\mathcal{I} \cup \{\mathsf{Ind}_G\}$ *w.r.t. the clauses* $l \simeq r$ *and* $\overline{L}[t[l\theta \mapsto r\theta]] \vee C$*.*

Lemmas 1–2 allow us to check for redundant Ind_G inferences similarly as performing demodulation, i.e. simplification by downward rewrites [18]. A consequence of these lemmas is that any non-redundant Ind_G inference on a clause that could be simplified by demodulation must induct on a subterm of a demodulatable term (i.e. a term that could be downward rewritten into a smaller term). The converse, however, that every subterm of a demodulatable term gives rise to a non-redundant Ind_G inference, does not hold in general.

Example 3. *As shown in Example 2, the* Ind_G *inference on clause* (2) *using induction term* c *is redundant w.r.t. clause* (ax.3) *and* (nConj) *due to* $[\![f(g(c))]\!] \succ \mathsf{s}([\![c]\!])$*. The only non-redundant* Ind_G *inferences on clause* (2) *w.r.t. clause* (ax.3) *thus induct on the subterms* $f(g(c))$ *or* $g(c)$ *of* $[\![f(g(c))]\!]$. □

To control over which clauses induction should be triggered, we introduce the following notion for clauses that are not directly usable as premises for induction inferences.

Definition 3 (Inductively redundant clause)**.** A clause is *inductively redundant* if it is (first-order) redundant and all Ind_G inferences on it are redundant.

The following example shows an inductively redundant clause.

Example 4. *Continuing Example 3, the term* $[\![g(d) \mathbin{@} f(c)]\!]$ *in clause* (2) *can be demodulated, namely into* $[\![f(d) \mathbin{@} g(c)]\!]$ *by clause* (ax.1)*. This makes inducting only on the subterms* $g(d)$*,* $f(c)$ *or* $g(d) \mathbin{@} f(c)$ *non-redundant. As* $[\![f(g(c))]\!]$ *and* $[\![g(d) \mathbin{@} f(c)]\!]$ *render all* Ind_G *inferences on the subterms of each other redundant, clause* (2) *is inductively redundant.* □

An inductively redundant clause is only necessary to preserve the ED property. Next, we show how some inductively redundant clauses can be avoided without losing ED. Towards this, we define so-called *ineffective* equations.

Definition 4 (Ineffective equation)**.** An equation $l \simeq r$ is *ineffective* if $l \succ r$, each variable in l has at most one occurrence and there is no strict non-variable subterm s in l s.t. $s\theta \in \mathcal{I}nd(\mathcal{T})$ for some substitution θ. An equation is called *effective* if it is not ineffective. We call an upward rewrite with an ineffective equation an *ineffective rewrite*.

The following example shows ineffective equations.

Example 5. *As discussed in Remark 2, base constructors such as* $\mathsf{0}$ *and* nil *are not inducted upon when using only constructor-based induction schemas; hence,* $\mathsf{0}, \mathsf{nil} \notin \mathcal{I}nd(\mathcal{T})$*. The equations* (plus.1)*,* (append.1) *and* (length.1) *are therefore ineffective, since they are oriented left-to-right, their left-hand sides are linear, and none of the strict non-variable subterms in their left-hand sides are inducted upon.* □

The following lemma proves that the result of an ineffective rewrite is inductively redundant.

Lemma 3 (Redundancy of ineffective rewrites)**.** *Let* $l \simeq r$ *be an ineffective equation and* $C[l\theta]$ *a clause. If* $C[l\theta] \succ (l \simeq r)\theta$*, the clause* $C[l\theta]$ *is inductively redundant in any inference system that admits ED.*

Consider a derivation of Rw inferences from an inductively non-redundant clause C into an inductively non-redundant conclusion D, where every intermediate clause in the derivation is inductively redundant. We may notice in such derivations that an ineffective rewrite is eventually followed by a rewrite that is not ineffective in an overlapping position. These rewrites can be performed together to avoid the intermediate inductively redundant clauses. To control and trigger such rewriting chains, we introduce the following *chaining* inferences.

$$(\mathsf{CRw})\ \frac{C[l\theta] \quad l \simeq r}{C[r\theta]} \qquad \text{where} \quad l\theta \not\prec r\theta \text{ or } l \simeq r \text{ is effective,}$$

$$(\mathsf{Chain}_1)\ \frac{s[l'] \simeq t \quad l \simeq r}{(s[r] \simeq t)\theta} \qquad \text{where} \quad \begin{array}{l}(1)\ \theta = \mathsf{mgu}(l, l'),\\ (2)\ s[l'] \simeq t \text{ is ineffective,}\\ (3)\ l\theta \not\succeq r\theta \text{ and } l \simeq r \text{ is effective,}\end{array}$$

$$(\mathsf{Chain}_2)\ \frac{s \simeq t[l'] \quad l \simeq r}{(s \simeq t[r])\theta} \qquad \text{where} \quad \begin{array}{l}(1)\ \theta = \mathsf{mgu}(l, l'),\\ (2)\ l \simeq r \text{ is ineffective,}\\ (3)\ t[l']\theta \not\succeq s\theta \text{ and } s \simeq t[l'] \text{ is effective.}\end{array}$$

The chaining inferences Chain_1 and Chain_2 combine ineffective and effective equations together in new, effective equations. Further, CRw disallows ineffective rewrites for consequence generation. By using chaining inferences for efficient rewrites in saturation with induction, we define the calculus **CReC** as $\mathbf{Sup} \cup \{\mathsf{CRw}, \mathsf{Chain}_1, \mathsf{Chain}_2\}$. While **CReC** does not admit ED, we note that if an inductively non-redundant clause D is derivable via Rw in **ReC**, then D is also derivable using chaining inferences in **CReC**. That is, inductive consequences are not lost in **CReC**. The following theorem adjusts such a variant of ED to **CReC**.

Theorem 5 (**CReC** derivability)**.** *Let D be an inductively non-redundant clause. If $\mathcal{C} \vdash^*_{\{\mathsf{Rw}\}} D$, then $\mathcal{C} \vdash^*_{\{\mathsf{CRw}, \mathsf{Chain}_1, \mathsf{Chain}_2\}} D$.*

The calculi $\mathbf{CReC}_\vee$, $\mathbf{CReC}^\rightarrow$ and $\mathbf{CReC}^\rightarrow_\vee$ are respectively the variants of $\mathbf{ReC}_\vee$, $\mathbf{ReC}^\rightarrow$ and $\mathbf{ReC}^\rightarrow_\vee$, when using Chain_1 and Chain_2, and restricting the Rw variants in these calculi to be used with effective equations similarly as in CRw.[5] Derivability results for these calculi similar to Theorem 5 are straightforward based on Theorems 2–5.

6 Evaluation

Implementation. We implement our calculi in the VAMPIRE[6] prover. Our framework for equational consequence generation is controlled via the new option `-grw` which has the following values: `off` disables equational consequence generation and uses only the **Sup** calculus; `all` uses the calculus **ReC**; `up` uses $\mathbf{ReC}_\vee$; `ltr` uses $\mathbf{ReC}^\rightarrow$; and `up_ltr` uses $\mathbf{ReC}^\rightarrow_\vee$. With the further option `-mgrwd`, we limit the maximum depth of rewrites for Rw inference variants. The option `-grwc` toggles the chaining inferences, that is, the use of **CReC** variants. Finally, the option `-indrc` controls the redundancy check for induction, by using Lemmas 1–2 to avoid performing redundant Ind_G inferences.

Additionally, we use the following heuristics to control consequence generation in saturation with induction. We apply rewriting in a goal-oriented manner, only allowing rewriting into conjectures or into clauses derived from conjectures (subgoals). Moreover, to avoid useless clauses from rewriting between unrelated subgoals, we disallow rewriting inferences which would introduce new Skolem constants into a clause. Finally, we avoid rewriting inferences which introduce variables into our conjectures, as these have to be instantiated before induction.

Experimental setup. We run our experiments with the following option setup: `-sa discount` to use the DISCOUNT saturation algorithm [13]; `-drc encompass` to enable encompassment

[5]See Appendix C of our extended paper [22].

[6]https://github.com/vprover/vampire/commit/16a38442515f8385

`-indrc off`	**Sup**	**ReC**	**ReC**$_\vee$	**ReC**$^\rightarrow$	**ReC**$^\rightarrow_\vee$	**CReC**	**CReC**$_\vee$	**CReC**$^\rightarrow$	**CReC**$^\rightarrow_\vee$
KBO	270	290	290	290	**291**	302	**305**	302	302
LPO	290	318	320	**322**	320	331	**332**	**332**	**332**

`-indrc on`	**Sup**	**ReC**	**ReC**$_\vee$	**ReC**$^\rightarrow$	**ReC**$^\rightarrow_\vee$	**CReC**	**CReC**$_\vee$	**CReC**$^\rightarrow$	**CReC**$^\rightarrow_\vee$
KBO	270	299	299	**301**	300	305	304	**307**	**307**
LPO	290	321	319	**322**	321	**333**	**333**	330	329

Figure 6: Comparison of the **Sup** calculus with variants of the **ReC** and **CReC** calculi, using 1266 inductive benchmarks. Redundant Ind_G inference detection is disabled (resp. enabled) in the top (resp. bottom) table with option `-indrc off` (resp. `-indrc on`). Maximum rewriting depth (`-mgrwd`) is set to 3. Bold entries show the maximum number of solved benchmarks among each subgroup.

demodulation [15]; and `-thsq on` to control pure theory derivations [19]. Experiments were run on computers with AMD Epyc 7502 2.5GHz processors and 1TB RAM, with each individual benchmark run given a single core. For inductive reasoning experiments, we used the UFDTLIA benchmark set from SMT-LIB [4], the TIP benchmark set [9] and the Vampire inductive benchmark set [21]. We also used benchmarks from the UEQ division of TPTP [36] to test first-order reasoning.

Evaluation of inductive reasoning. The first part of our experiments consisted of running Vampire on *1266 inductive benchmarks* from the UFDTLIA, TIP and Vampire benchmark sets. We used a 60-second timeout and the options `-ind struct -indoct on` to enable induction and generalisations over complex terms. We used two different simplification orderings: `-to kbo` for KBO ordering with constant weight and precedence determined by the arity of symbols; `-to lpo -sp occurrence` for the LPO ordering with a symbol precedence given by the declaration order. This LPO order is usually better at orienting recursive function axioms [23].

Our results are summarised in Figure 6, showcasing that each **ReC** and **CReC** calculi variant performs significantly better than **Sup**. Using the LPO ordering turned out to be advantageous over the KBO ordering. Performance is further improved via detection of redundant Ind_G inferences and using chaining inferences via **CReC** variants. Among each group, however, the differences in the number of solved benchmarks are minimal, and there is no calculus that is a clear winner in all configurations. Statistics reveal that redundant Ind_G inference detection used together with **ReC** was able to eliminate 79.3% of the overall 390,657,294 Ind_G inferences, while redundant Ind_G inference detection in **CReC** variant runs detected 42.6% of the overall 169,457,851 Ind_G inferences redundant. This suggests that both redundant Ind_G detection and chaining inferences in **CReC** variants are effective in keeping the search space small. In total, the configurations for the **ReC** and **CReC** variants solved 45 problems that no **Sup** variant could solve. Based on these results, we conclude that rewriting in inductive reasoning significantly improves upon standard superposition.

Figure 7 shows cactus plots [7] within the LPO configuration of **Sup**, **ReC** and **CReC** variants, and **ReC** and **CReC** variants with redundant Ind_G inference detection. Each plot line lists the logarithm of the time needed (vertical axis) to solve a certain number of the benchmarks (horizontal axis) individually, for a particular configuration. The left diagram shows the entire plot, and the right diagram a magnified (and rescaled) plot above 180 problems. The baseline configuration **Sup** is a bit faster than the **ReC** variants up to around 260 problems, and after that it only solves a few problems in the several seconds region. The **ReC** variants without

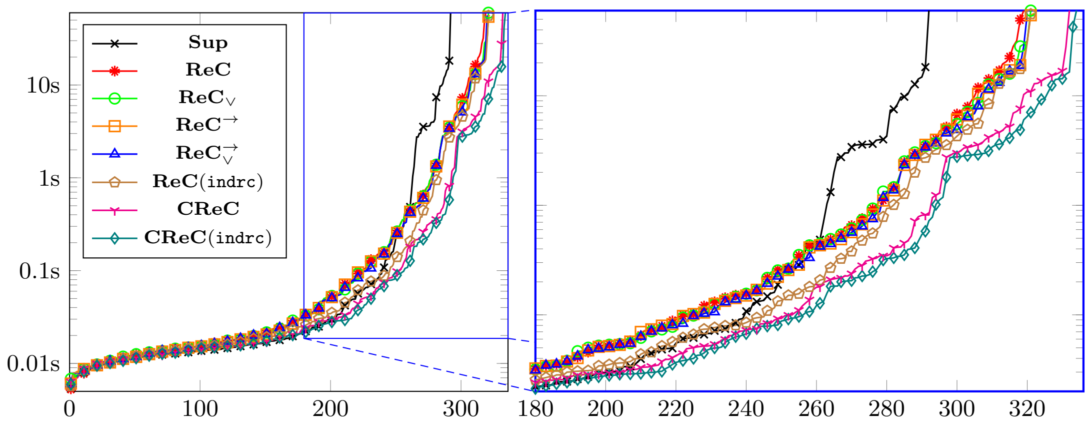

Figure 7: Plots of (logarithmic) time against number of problems that would be solved individually given that time limit. Selected configurations, all using LPO.

redundant Ind_G detection are almost indistinguishable in the entire plot. The **ReC** calculus with redundant Ind_G detection is however better, while there is a greater gap between the **Sup** and **ReC** calculi and the two **CReC** variants. The **CReC** calculus with redundant Ind_G detection has the slowest growing curve, corresponding to the fastest solving times.

Evaluation of first-order reasoning. We also measured how our calculi behave with pure first-order problems. In particular, we have run experiments on the UEQ division of TPTP using the CASC2019 portfolio mode of VAMPIRE with a 300 seconds timeout (`--mode portfolio -sched casc_2019 -t 300`). While our calculi performed slightly worse than VAMPIRE portfolio, they managed to solve a few unique and hard UEQ problems VAMPIRE without consequence generation could not solve: `GRP664-12` (rating 0.96), `COL066-1` (rating 0.79), `LAT166-1` (rating 0.71), `LAT156-1` (rating 0.71) and `REL026-1` (rating 0.67). As such, our work is useful not only for inductive, but also for first-order reasoning.

7 Related Work and Conclusion

We improve the generation of equational consequences within saturation-based theorem proving extended with inductive reasoning. The generated consequences serve as auxiliary lemmas to be used for proving (inductive) goals.

While auxiliary lemmas might be provided by users in interactive theorem proving [14, 31], saturation-based automated theorem provers, by design, do not support user guidance during proof search. Automation of induction in saturation therefore implements inductive generalizations [1, 12, 24, 33], uses failed proof attempts [8], specialized sound inferences with common patterns [38], or integrates induction directly into saturation [16,27]. Our work extends these techniques by guiding proof search with auxiliary lemmas generated during proof search. Lemma generation is also exploited in theory exploration [10, 25], without however imposing the relevance of generated lemmas with respect to a given conjecture. SMT solvers alleviate this by integrating theory exploration with built-in theory reasoning [34].

The use of equational theories and term rewriting in inductive reasoning has been addressed in [6], by ensuring termination of function definitions, and in [23,38], by using using completion procedures [5,8] to interleave heuristic rewriting with theorem proving. Our work complements these approaches, by using rewriting-based equational reasoning for auxiliary lemma generation, and extending redundancy elimination in inductive reasoning. Our experiments have shown a significant improvement in inductive reasoning and equational first-order reasoning, solving 45 new inductive problems and 5 hard TPTP problems when compared with standard superposition.

Integrating our method with proof assistants, e.g. Sledgehammer [14], is an interesting line of future work, with the aim of splitting goals into subgoals described by auxiliary lemmas. Applying our approach to non-equational fragments, such as Horn formulas in equational logic [11], is another future challenge.

Acknowledgements. We thank Martin Suda for discussing ideas related to this work. This work was partially funded by the ERC Consolidator Grant ARTIST 101002685, the TU Wien Doctoral College SecInt, the FWF SFB project SpyCoDe F8504, the WWTF ICT22-007 grant ForSmart, and the Amazon Research Award 2023 QuAT.

References

[1] Raymond Aubin. *Mechanizing Structural Induction.* PhD thesis, University of Edinburgh, UK, 1976.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, USA, 1998.

[3] Leo Bachmair and Harald Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In *Automated Deduction: A Basis for Applications.* Kluwer, 1998.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[5] Adel Bouhoula and Michael Rusinowitch. Implicit Induction in Conditional Theories. *Journal of Automated Reasoning*, 1995.

[6] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[7] Martin Brain, James H. Davenport, and Alberto Griggio. Benchmarking Solvers, SAT-style. In *Proceedings of the 2nd SCSC workshop 2017*, CEUR Workshop Proceedings, 2017.

[8] A. Bundy, A. Stevens, F. V. Harmelen, A. Ireland, and A. Smaill. Rippling: A Heuristic for Guiding Inductive Proofs. *Artificial Intelligence*, 1993.

[9] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: Tons of Inductive Problems. In *Intelligent Computer Mathematics*, 2015.

[10] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating Inductive Proofs Using Theory Exploration. In *CADE*, 2013.

[11] Koen Claessen and Nicholas Smallbone. Efficient Encodings of First-Order Horn Formulas in Equational Logic. In *IJCAR*, 2018.

[12] Simon Cruanes. Superposition with Structural Induction. In *FroCoS*, 2017.

[13] Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. DISCOUNT - A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 1997.

[14] Martin Desharnais, Petar Vukmirović, Jasmin Blanchette, and Makarius Wenzel. Seventeen Provers Under the Hammer. In *ITP*, 2022.

[15] André Duarte and Konstantin Korovin. Ground Joinability and Connectedness in the Superposition Calculus. In *IJCAR*, 2022.

[16] M. Echenheim and N. Peltier. Combining Induction and Saturation-Based Theorem Proving. *Journal of Automated Reasoning*, 2020.

[17] Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, and Giles Reger. The Rapid Software Verification Framework. In *FMCAD*, 2022.

[18] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption demodulation in first-order theorem proving. In *IJCAR*, 2020.

[19] Bernhard Gleiss and Martin Suda. Layered Clause Selection for Saturation-Based Theorem Proving. In *PAAR and SC-Square @ IJCAR 2020, Proceedings*, 2020.

[20] Márton Hajdu, Petra Hozzová, Laura Kovács, Giles Reger, and Andrei Voronkov. Getting Saturated with Induction. In *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, 2022.

[21] Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with Generalization in Superposition Reasoning. In *CICM*, 2020.

[22] Márton Hajdú, Laura Kovács, and Michael Rawson. Rewriting and Inductive Reasoning (Extended Version). *CoRR*, abs/2402.19199, 2024.

[23] Márton Hajdu, Petra Hozzová, Laura Kovács, and Andrei Voronkov. Induction with Recursive Definitions in Superposition. In *FMCAD*, 2021.

[24] Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer Induction in Saturation. In *CADE*, 2021.

[25] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture Synthesis for Inductive Theories. *Journal of Automated Reasoning*, 2011.

[26] Deepak Kapur and Hantao Zhang. RRL: A rewrite rule laboratory. In *CADE*, 1988.

[27] A. Kersani and N. Peltier. Combining Superposition and Induction: A Practical Realization. In *FroCoS*, 2013.

[28] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, 2013.

[29] Shuvendu Lahiri, Akash Lal, Yi Li, and Ankush Das. Angelic Verification: Precise Verification Modulo Unknowns. In *CAV*, 2015.

[30] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*. MIT Press, 2001.

[31] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Berlin Heidelberg, 2002.

[32] Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. Specification synthesis with constrained Horn clauses. In *PLDI*, 2021.

[33] Giles Reger and Andrei Voronkov. Induction in Saturation-Based Proof Search. In *CADE*, 2019.

[34] Andrew Reynolds and Viktor Kuncak. Induction for SMT Solvers. In *VMCAI*, 2015.

[35] Rolf Socher-Ambrosius. A goal oriented strategy based on completion. In *Algebraic and Logic Programming*, 1992.

[36] G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022.

[37] Christoph Walther. Computing induction axioms. In *LPAR*, 1992.

[38] Daniel Wand. *Superposition: Types and Induction*. PhD thesis, Saarland University, Saarbrücken, Germany, 2017.