HAL
open science

Combining Superposition and Induction: A Practical Realization

Abdelkader Kersani, Nicolas Peltier

To cite this version:

Abdelkader Kersani, Nicolas Peltier. Combining Superposition and Induction: A Practical Realization. FroCoS 2013 - 9th International Symposium on s of Combining Systems (co-located with TABLEAUX 2013), Sep 2013, Nancy, France. pp.7-22, 10.1007/978-3-642-40885-4_2 . hal-00934610

HAL Id: hal-00934610

https://hal.science/hal-00934610v1

Submitted on 22 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Combining Superposition and Induction: a Practical Realization*

Abdelkader Kersani and Nicolas Peltier

University of Grenoble (LIG, CNRS)

.

Abstract. We consider a proof procedure aiming at refuting clause sets containing arithmetic constants (or parameters), interpreted as natural numbers. The superposition calculus is enriched with a loop detection rule encoding a form of mathematical induction on the natural numbers (by "descente infinie"). This calculus and its theoretical properties are described in [2, 16]. In the present paper, we focus on more practical aspects. We provide algorithms to apply the loop detection rule in an automatic and efficient way. We describe a research prototype implementing our technique and provide some preliminary experimental results.

1 Introduction

We consider first-order formulæ built on a language containing constant symbols interpreted as natural numbers. As an example, consider the formula ϕ defined as the conjunction of the following formulæ:

$$p(0, a)$$
$$\forall x, y \neg p(x, y) \vee p(x + 1, f(y))$$
$$\exists n \forall x \neg p(n, x)$$

The formula ϕ is satisfiable in the usual sense, but it is unsatisfiable if the sort of the first argument of p is interpreted as the natural numbers (with the usual interpretation of $0, 1$ and $+$). Then the existential variable n must be interpreted as a natural number k, and it is easy to check, by induction on k, that the first two formulæ entail that $p(k, f^k(a))$ holds, which implies that the formula is unsatisfiable. Existing resolution or superposition based theorem provers cannot establish the unsatisfiability of such formulæ since they are based on standard first-order logic. Proof procedures (based on several different approaches) have been proposed to handle hybrid formulæ, mixing first-order logic with interpreted theories such as Presburger arithmetic [4, 1, 6, 12] but they do not handle inductive theorems. When fed with the previous formula, these approaches will infer the infinite set of formulæ $n \not\simeq 0$, $n \not\simeq 1$, $n \not\simeq 2$, ... (where n denotes the Skolem constant

* This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

derived from the quantification $\exists n$), but will not detect unsatisfiability in finite time (since Presburger arithmetic is not compact). The standard approach for dealing with inductive theorems in the context of first-order theorem proving is to add explicit induction schemes. For instance, in the previous case, one can replace the formula $\exists n \forall x \neg p(n,x)$ by $\exists n \forall x \neg p(n,x) \wedge \forall m\, m+1 \not\simeq n \vee \exists x\, p(m,x)$ (stating that $\forall x \neg p(m,x)$ holds for $m = n$ but not for the predecessor of n) which can be easily derived by assuming that n is the *minimal* natural number satisfying the property $\forall x \neg p(n,x)$. Alternatively, one can also add the usual induction scheme using $\exists x\, p(m,x)$ as an inductive invariant:

$$(\exists x\, p(0,x) \wedge \forall m((\exists x\, p(m,x)) \Rightarrow (\exists x\, p(m+1,x)))) \Rightarrow \forall m \exists x\, p(m,x)$$

Using these additional axioms, the unsatisfiability of ϕ can easily be established by any theorem prover. The inductive rule defined in [15] also relies on the use of explicit induction schemes.

However, this approach relies on the user to guess the right inductive lemma. The inductive invariant is not necessarily equivalent to the goal, and is not even bound to occur in the initial formula (it is well-known that inductive proofs do not admit cut elimination). For instance, if the third formula is replaced by: $\exists n \forall x \neg q(n,x)$ with the additional axiom: $\forall x, y\, q(x,y) \vee \neg p(x,y)$, then the formula cannot be established by using the negation of the goal $\exists x\, q(n,x)$ as an inductive invariant: one has to use $\exists x\, p(n,x)$ instead.

Another approach consists in using inductive theorem provers, which are usually based on rewriting [7, 13, 14, 20, 11]. These approaches allow one to generate automatically the induction lemmata (in some cases). Intuitively, these procedures work as follows: the goal is rewritten using axioms until it can be reduced to `true` or `false`. Of course only the ground instances of the goal can be normalized and enumerating those instances does not terminate in general. In order to ensure termination in some cases, the previously encountered goals can themselves be used as derived rewrite rules, provided the considered terms are strictly lower than the initial ones, according to some reduction ordering, which can be either fixed a priori or constructed dynamically all along the search. This technique allows one to simulate the application of inductive hypotheses without having to state explicitly the inductive invariants (of course additional inductive lemmata still have to be added by hand in many cases). However, these approaches are restricted to goals of the form $\forall \boldsymbol{x}\, \psi$ where ψ is a quantifier-free formulæ, thus they cannot handle formulæ as ϕ in the previous example, whose goal is of the form $\forall n \exists x\, \psi$, before negation (the inductive theorem prover SPIKE has been extended in order to handle existential variables [5], but the use of such variables is strongly restricted). The "inductionless induction" approach [8], which reduces inductive theorem proving to a consistency test in first-order logic, suffers from the same limitation.

In previous work [2, 16], we have presented an extension of the superposition calculus which is tailored to handle formulæ such as the previous one. The idea is twofold. First, the arithmetic terms are abstracted away and replaced by variables, in order to allow inferences on them. This allows one to get rid

of first-order symbols in order to derive properties of pure arithmetic terms. Second, the usual inference rules of the calculus are enriched with a new rule allowing to detect cycles in the derivations. These loops correspond to the inductive invariants that are needed to establish the validity of the theorem. A first definition of the loop detection rule is given in [2] and a more general version is provided in [16] yielding stronger completeness results (of course the method is not complete in general, since the logic is not even semi-decidable [16]). Roughly speaking, these rules apply when a set of clauses $S[n]$ is generated (where n is an arithmetic constant) such that the set $n \geq k \wedge S[n-k]$ can be derived from $S[n]$ (using the inference rules of the calculus). By descente infinie, it is clear that this implies that S is unsatisfiable. The soundness of this approach is proven in [2, 16], and some (partial) completeness results are presented. In the present paper, we tackle more practical aspects, namely the efficient generation of the sets of clauses on which the loop detection rule can be applied. The problem consists in finding efficiently sets of clauses S satisfying the relation above. We present two different algorithms for performing this task (each with their pros and cons). We describe an implementation of our method and provide some preliminary experimental results.

2 Syntax and Semantics

We firstly define the syntax and semantics of the considered logic. We assume some familiarity with the usual notions in logic and automated deduction (missing definitions can be found in, e.g., [18]). We consider two distinct sorts: the sort `term` of the standard terms, and the sort `nat` of the natural numbers. The set of terms is built as usual on a set of function symbols Σ and on a set of variables $\mathcal{X}$. The signature Σ contains in particular the symbols 0 and *succ*, of profile `nat` and `nat` $\rightarrow$ `nat` respectively. We assume that Σ contains no other symbol of range `nat`.

An *atom* is an equation of the form $t \simeq s$ where t and s are terms of sort `term`. A *literal* is either an atom (positive literal) or the negation of an atom (negative literal). A *clause* is a finite set (written as a disjunction) of literals. Let n be a special symbol, called the *parameter*, not occurring in Σ (n is intended to denote a natural number, and can be viewed as a constant symbol of sort `nat`).

Definition 1. *An* n-clause *is a pair* $[C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ *where* C *is a clause and the* t_i*'s* $(1 \leq i \leq k)$ *are terms of sort* `nat`*. It is* normalized *if* $k \in \{0, 1\}$*. If* $k = 0$*,* $\bigwedge_{i=1}^{k} n \simeq t_i$ *is equivalent to* `true` *by convention and* $[C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ *is simply written* C*.* C *is the* clausal part *of the* n*-clause, and* $\bigwedge_{i=1}^{k} n \simeq t_i$ *is the* constraint*.*

Note that by definition, n can only occur in the constraint part of the n-clause (since $n \notin \Sigma$). Thus an expression of the form $f(n) \simeq a$ for instance is to be written as $[f(x) \simeq a \mid n \simeq x]$, where x is a variable of sort `nat`.

For every expression e, $\mathrm{var}(e)$ denotes the set of variables occurring in e. An expression is *ground* if it contains no variable.

A *substitution* σ is a function mapping every variable to a term of the same sort. The *domain* of σ is the set of variables x such that $x\sigma \neq x$. For every expression e, $e\sigma$ denotes as usual the expression obtained from e by replacing every occurrence of each variable x by $x\sigma$. The substitution σ is *ground* iff for every variable x in the domain of σ, $x\sigma$ is ground.

The terms $t_1, \ldots, t_k$ are *unifiable* iff there exists a substitution σ such that $t_1\sigma = \cdots = t_k\sigma$. Any set of unifiable terms has a most general unifier (unique up to a renaming).

We identify a term $succ^k(0)$ with the natural number k; thus we write, e.g., $succ^k(0) < succ^l(0)$ for $k < l$, or $k + t$ for $succ^k(t)$.

Interpretations are usually defined as congruences on the set of terms. In our setting, we also have to specify the value of the symbol n (the symbols 0 and *succ* are interpreted as free constructors, note that the clauses contain no equations between natural numbers). This yields the following:

Definition 2. *An* interpretation $\mathcal{I}$ *is defined by a pair* $(n_\mathcal{I}, \simeq_\mathcal{I})$*, where* $n_\mathcal{I}$ *is natural number (i.e., a term of the form* $succ^k(0)$*) and* $\simeq_\mathcal{I}$ *is a congruence on the set of ground terms of sort* `term`*.*

The notion of validity is defined in a very natural way:

Definition 3. *An interpretation* $\mathcal{I}$ validates *an expression* E *(written* $\mathcal{I} \models E$*) iff one of the following conditions holds.*

- E *is a ground literal* $t \simeq s$ *(resp.* $t \not\simeq s$*) and* $t \simeq_\mathcal{I} s$ *(resp.* $t \not\simeq_\mathcal{I} s$*).*
- E *is a ground clause* $\bigvee_{i=1}^k l_i$ *and there exists* $i \in [1, k]$ *such that* $\mathcal{I} \models l_i$*.*
- E *is an* n*-clause* $[C \mid \bigwedge_{i=1}^k n \simeq t_i]$ *and for every ground substitution* σ *of domain* $var(E)$ *such that* $\forall i \in [1, k], n_\mathcal{I} = t_i\sigma$*, it holds that* $\mathcal{I} \models C\sigma$*.*
- E *is a set of* n*-clauses and* $\forall \mathcal{C} \in E, \mathcal{I} \models \mathcal{C}$*.*

An interpretation validating E *is a* model *of* E*. We write* $E \models E'$ *if every model of* E *is a model* E'*. Two expressions* E *and* E' *are* equivalent *(written* $E \equiv E'$*) if* $E \models E'$ *and* $E' \models E$*. A* tautology *is an expression equivalent to* `true`*.*

By definition, $\mathcal{I} \models [\Box \mid n \simeq succ^k(0)]$ iff $n_\mathcal{I} \neq k$. Similarly, $\mathcal{I} \models [\Box \mid n \simeq succ^k(x)]$ iff $n_\mathcal{I} < k$ (where x is a variable). Consequently, an n-clause of the form $[\Box \mid n \simeq succ^k(0)]$ (resp. $[\Box \mid n \simeq succ^k(x)]$) will be written $n \not\simeq k$ (resp. $n < k$).

If $\mathcal{I}$ is an interpretation and k is a natural number, we denote by $\mathcal{I}[k/n]$ the interpretation coinciding with $\mathcal{I}$, except that the value of n is set to k.

The following proposition shows that every non-tautological n-clause is equivalent to a normalized n-clause.

Proposition 1. *Let* $\mathcal{C} = [C \mid \bigwedge_{i=1}^k n \simeq t_i]$ *be an* n*-clause. If* $t_1, \ldots, t_n$ *are unifiable, then* $\mathcal{C}$ *is equivalent to* $[C\sigma \mid n \simeq t_1\sigma]$*, where* σ *is an m.g.u. of* $t_1, \ldots, t_n$*. Otherwise* $\mathcal{C}$ *is a tautology.*

Thanks to Proposition 1 we can safely assume that every n-clause is normalized (the normalization operation is applied in a systematic way to every generated n-clause).

The usual relation of subsumption extends straightforwardly to n-clauses:

Definition 4. *Let* $\mathcal{C} = [C \mid \bigwedge_{i=1}^{k} n \simeq t_i]$ *and* $\mathcal{C}' = [C' \mid \bigwedge_{i=1}^{l} n \simeq t'_i]$ *be two* n*-clauses. The* n*-clause* $\mathcal{C}$ subsumes $\mathcal{C}'$ *(written* $\mathcal{C} \leq_{sub} \mathcal{C}'$*) if there exists a substitution* σ *such that* $C\sigma \subseteq C'$ *and* $\{t_1, \ldots, t_k\}\sigma \subseteq \{t'_1, \ldots, t'_l\}\sigma$.

Proposition 2. *If* $\mathcal{C} \leq_{sub} \mathcal{C}'$ *then* $\mathcal{C} \models \mathcal{C}'$.

The subsumption relation $\leq_{sub}$ can be extended to sets of n-clauses: we write $S \leq_{sub} S'$ if for every $\mathcal{C}' \in S'$, there exists $\mathcal{C} \in S$ such that $\mathcal{C} \leq_{sub} \mathcal{C}'$.

By Definition 3, an n-clause $[C \mid n \simeq succ^i(x)]$ (with $x \in \mathcal{X}$) is equivalent to an expression of the form $n \geq i \Rightarrow C[n-i]$. The *rank* of $[C \mid n \simeq succ^i(x)]$ is the number r such that $n - r$ is the maximal expression containing n occurring in $C[n-i]$. For instance, consider the n-clause $[f(x, succ(y)) \simeq y \mid n \simeq succ^i(x)]$. It is equivalent to the expression $f(x, succ(n-i)) \simeq n-i$, i.e., $f(x, n-(i-1)) \simeq n-i$, hence its rank is $i - 1$. Note that if C contains no occurrence of *succ* then the rank of $[C \mid n \simeq succ^i(x)]$ is simply i. For every set of n-clauses S and for every natural number i, we denote by $S[i]$ the set of n-clauses of rank i in S. We denote by $S[\top]$ the set of n-clauses whose constraint is `true`.

3 Superposition Calculus

The usual superposition calculus can easily be extended to operate on n-clauses. Let $<$ be a reduction ordering and let *sel* be a selection function, mapping every clause C to a subset of C, such that either $sel(C)$ contains a negative literal, or $sel(C)$ contains all the $<$-maximal literals in C. The calculus is defined by the following three rules (the premisses are assumed to be normalized). As usual $t|_p$ is the term occurring at position p in t, and $t[s]_p$ is the term obtained from t by replacing the subterm at position p by s.

Superposition

$$\frac{[C \vee t \bowtie s \mid \mathcal{X}], [D \vee u \simeq v \mid \mathcal{Y}]}{[C \vee D \vee t[v]_p \bowtie s \mid \mathcal{X} \wedge \mathcal{Y}]\sigma}$$

If $\bowtie \in \{\simeq, \not\simeq\}$, $\sigma = \mathrm{mgu}(u, t|_p)$, $u\sigma \not\leq v\sigma, t\sigma \not\leq s\sigma$, $t|_p$ is not a variable, $(t \bowtie s)\sigma \in sel((C \vee t \bowtie s)\sigma)$, $(u \simeq v)\sigma \in sel((D \vee u \simeq v)\sigma)$.

Reflection

$$\frac{[C \vee t \not\simeq s \mid \mathcal{X}]}{[C \mid \mathcal{X}]\sigma} \qquad \text{If } \sigma = \mathrm{mgu}(t, s),\ (t \not\simeq s)\sigma \in sel((C \vee t \not\simeq s)\sigma)$$

Factorisation

$$\frac{[C \vee t \simeq s \vee u \simeq v \mid \mathcal{X}]}{[C \vee s \not\simeq v \vee t \simeq s \mid \mathcal{X}]\sigma}$$

If $\sigma = \mathrm{mgu}(t, u)$, $t\sigma \not< s\sigma$, $u\sigma \not< v\sigma$, $(t \simeq s)\sigma \in sel((C \vee t \simeq s \vee u \simeq v)\sigma)$.

Example 1. The second example of the Introduction is formalized as follows.

$$\begin{array}{ll} 1 & p(0,a) \simeq \texttt{true} \\ 2 & p(x,y) \not\simeq \texttt{true} \vee p(succ(x), f(y)) \simeq \texttt{true} \\ 3 & [q(x,y) \not\simeq \texttt{true} \mid n \simeq x] \\ 4 & q(x,y) \simeq \texttt{true} \vee p(x,y) \not\simeq \texttt{true} \end{array}$$

The following n-clauses are generated (for the sake of clarity, the literals of the form $\texttt{true} \not\simeq \texttt{true}$ are removed from the clauses):

$$\begin{array}{lll} 5 & [p(x,y) \not\simeq \texttt{true} \mid n \simeq x] & \text{(superposition, 4, 3)} \\ 6 & [\square \mid n \simeq 0] & \text{(superposition, 1, 5)} \\ 7 & [p(x,y) \not\simeq \texttt{true} \mid n \simeq succ(x)] & \text{(superposition, 2, 5)} \\ 8 & [\square \mid n \simeq succ(0)] & \text{(superposition, 1, 7)} \\ 9 & [p(x,y) \not\simeq \texttt{true} \mid n \simeq succ(succ(x))] & \text{(superposition, 2, 7)} \end{array}$$

An infinite number of n-clauses of the form $[\square \mid n \simeq succ^k(0)]$ (i.e. $n \not\simeq k$) can be generated.

4 Loop Detection

In this section, we define the key part of the proof procedure, namely the loop detection rule. We provide a simpler and abstract version of the rule (compared with those in [2, 16]) which is sufficient for our purposes (the loop detection rule provided in [16] is much more general because it handles parameters interpreted as words instead of natural numbers). In the next section, we will introduce new definitions and algorithms allowing for an efficient application of the rule. We first need to introduce some notations.

Definition 5. *For every natural number i and for every n-clause $\mathcal{C} = [C \mid \bigwedge_{j=1}^{k} n \simeq t_j]$, we denote by $\mathcal{C}{\downarrow_i}$ the n-clause $[C \mid \bigwedge_{j=1}^{k} n \simeq succ^i(t_j)]$. If S is a set of n-clauses, then $S{\downarrow_i} \stackrel{def}{=} \{\mathcal{C}{\downarrow_i} \mid \mathcal{C} \in S\}$.*

Intuitively, $S{\downarrow_i}$ denotes the same formula as S, with n replaced by $n - i$. This yields the following:

Proposition 3. *Let $i, j \in \mathbb{N}$, let S be a set of n-clauses and let $\mathcal{I}$ be an interpretation. If $\mathcal{I} \models S{\downarrow_j}$ and $\mathcal{I}(n) = i + j$ then $\mathcal{I}[i/n] \models S$.*

Proof. Let $\mathcal{C} = [C \mid \bigwedge_{l=1}^{k} n \simeq t_l] \in S$. Let σ be a substitution such that $\forall l \in [1,k]\, n_{\mathcal{I}[i/n]} = t_i\sigma$. This implies that $\forall l \in [1,k], t_l\sigma = i$ since $n_{\mathcal{I}[i/n]} = i$ by definition. We have $\mathcal{C}{\downarrow_j} = [C \mid \bigwedge_{l=1}^{k} n \simeq succ^j(t_l)]$. Since $t_l\sigma = i$ and $n_{\mathcal{I}} = i + j$ we have $n_{\mathcal{I}} = t_l\sigma$, for all $l \in [1,k]$. Since $\mathcal{I} \models S{\downarrow_j}$, this entails that $\mathcal{I} \models C\sigma$, and thus $\mathcal{I}[i/n] \models C\sigma$ (since $C\sigma$ contains no occurrence of n).

Definition 6. *Let S be a set of clauses. A pair of natural numbers (i, j) (with $j \neq 0$) is an* inductive loop *for S if there exists a set $S' \subseteq S$ such that $S' \models n \not\simeq l$, for every $l \in [i, i+j[$ and $S' \models S'{\downarrow_j}$*

Theorem 1. *If (i, j) is an inductive loop for S, then $S \models n < i$.*

Proof. We have $\mathcal{I} \models S'$. Let k be the minimal natural number greater or equal to i such that S' has a model $\mathcal{I}$ with $n_{\mathcal{I}} = k$. If $k < i + j$, then we have $S' \models n \not\simeq k$, which is impossible since $\mathcal{I} \not\models n \not\simeq k$, by definition. Otherwise, we have $\mathcal{I} \models S'{\downarrow_j}$, hence by Proposition 3 we deduce that $\mathcal{I}[k - j/n] \models S'$, which is impossible by minimality of k since $i \leq k - j < k$.

5 Practical Application of the Loop Detection Rule

This section contains the main new results of the paper. We define algorithms to test whether a pair of natural numbers (i, j) is an inductive loop. To this purpose, we have to check whether there exists a set of n-clauses S' satisfying the conditions of Definition 6. In practice, these conditions cannot be tested since semantic entailment is undecidable. We will thus only check whether the formulæ $n \not\simeq l$ (with $i \leq l < i + j$) and $S'{\downarrow_j}$ can be derived from S' using inferences previously performed by the prover. Furthermore, we will impose the additional restriction that all the n-clauses in S' containing n occur in the set $S[i]$. This condition greatly decreases the search space and it preserves the completeness results in [2, 16].

We proceed in two steps. First we transform the semantic conditions of Definition 6 into purely syntactic properties and then we provide algorithms to test these properties in an effective way. We need to introduce additional notations.

Definition 7. *Let S be a set of n-clauses. An* inference relation *δ for S is a partial function mapping every n-clause $\mathcal{C} \in S$ to a set of n-clauses in S such that $\mathcal{C}$ is deducible from $\delta(\mathcal{C})$ by one of the inference rules (in exactly one step).*

In practice this relation will be obtained from the inferences previously performed by the prover ($\mathcal{D} \in \delta(\mathcal{C})$ iff $\mathcal{D}$ is a parent of $\mathcal{C}$). The n-clauses $\mathcal{C}$ such that $\delta(\mathcal{C})$ is not defined are hypotheses, i.e., n-clauses occurring in the initial set. An inference relation induces a entailment relation $\vdash_\delta$ between subsets of S. Informally, we write $S' \vdash_\delta S''$ if all the n-clauses in S'' can be derived from n-clauses in $S' \cup S[\top]$ using inferences in δ. This is formalized as follows:

Definition 8. *Let S be a set of n-clauses and let δ be an inference relation for S. The relation $\vdash_\delta$ is the smallest relation between subsets of S such that $S' \vdash_\delta S''$ if for every n-clause $\mathcal{C} \in S''$, one of the following conditions holds:*

1. *$\mathcal{C} \in S'$.*
2. *$\delta(\mathcal{C})$ is defined and $S' \vdash_\delta \delta(\mathcal{C})$.*
3. *The constraint part of $\mathcal{C}$ is* `true`*.*

The intuition behind Condition 3 is that the n-clauses whose constraints are `true` are universal properties, which are valid independently of the value of n. Thus we assume that such n-clauses (once they have been proven, i.e., if they occur in S) can be used as hypotheses in any derivation.

Proposition 4. *Let S be a set of n-clauses and let δ be an inference relation for S. If $S' \vdash_\delta S''$ then $S' \cup S[\top] \models S'' \cup S[\top]$.*

In order to test entailment between clause sets, we introduce a notion of immediate entailment:

Definition 9. *An* immediate entailment relation *is a decidable relation $\sqsupseteq$ between n-clauses such that $\mathcal{C} \sqsupseteq \mathcal{D} \Rightarrow \mathcal{C} \models \mathcal{D}$. The relation $\sqsupseteq$ is extended to sets of n-clauses as follows: $S \sqsupseteq S'$ iff $\forall \mathcal{C}' \in S' \, \exists \mathcal{C} \in S, \mathcal{C} \sqsupseteq \mathcal{C}'$.*

In practice, $\sqsupseteq$ can be for instance the identity ($\mathcal{C} = \mathcal{D}$ up to a renaming of variables), the inclusion ($\mathcal{C} \subseteq \mathcal{D}$), or the subsumption relation $\leq_{sub}$.

We are now in position to define the notion of cycle, which is the syntactic pendant of the notion of inductive loop of Definition 6. It is defined relatively to the two relations $\sqsupseteq$ and δ.

Definition 10. *Let S be a set of clauses, let $\sqsupseteq$ be an immediate consequence relation and let δ be an inference relation on S. A pair of natural numbers (i, j) (with $j \neq 0$) is a* cycle *for S w.r.t. $\sqsupseteq$ and δ if there exist $S_{init}, S_{loop} \subseteq S$ such that $S_{init} \subseteq S[i], S_{loop} \subseteq S[i+j]$, $S_{init} \vdash_\delta \{n \not\simeq k \mid k \in [i, i+j-1]\}$, $S_{init} \vdash_\delta S_{loop}$ and $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$.*

Example 2. Consider the derivation of Example 1. Assume that $\sqsupseteq$ is the identity relation. The pair $(0, 1)$ is a cycle, with $S_{init} = \{5\}$ and $S_{loop} = \{7\}$. Indeed, the clause $n \not\simeq 0$ is derivable from Clause 5 and clauses not containing n, thus $S_{init} \vdash_\delta \{n \not\simeq 0\}$. Similarly, Clause 7 can be derived from Clause 5, together with clauses not containing n. Finally, we have $[p(x, y) \not\simeq \texttt{true} \mid n \simeq x]{\downarrow_1} = [p(x, y) \not\simeq \texttt{true} \mid n \simeq succ(x)]$, hence $S_{loop} \sqsupseteq S_{init}{\downarrow_1}$ (assuming that $\sqsupseteq$ contains the identity relation). Similarly, $(0, 2)$ and $(1, 2)$ are also cycles, corresponding to the sets $\{5\}$, $\{9\}$ and $\{7\}$, $\{9\}$ respectively.

Theorem 2. *All cycles are inductive loops.*

Proof. Let (i, j) be a cycle for S, w.r.t. two relations $\sqsupseteq$ and δ. Let S_{init} and S_{loop} be the corresponding subsets of S. Let $S' = S_{init} \cup S[\top]$. We have $S_{init} \vdash_\delta \{n \not\simeq k\}$, for every $k \in [i, i+j-1]$, thus by Proposition 4, $S' \models n \not\simeq k$ (for every $k \in [i, i+j-1]$). Similarly, we have $S' \models S_{loop}$, hence $S' \models S_{init}{\downarrow_j}$ (since $\sqsupseteq$ is included in $\models$). But $S'{\downarrow_j} = S_{init}{\downarrow_j} \cup S[\top]{\downarrow_j} = S_{init}{\downarrow_j} \cup S[\top]$ (since the n-clauses whose constraint is `true` are not affected by the $S{\downarrow_j}$ operation). Therefore $S' \models S'{\downarrow_j}$ and (i, j) is an inductive loop.

Theorems 1 and 2 entail the soundness of the following inference rule:

$$\text{Cycle elimination rule:} \qquad \frac{S}{n < i}$$

If (i, j) is a cycle for S w.r.t. some immediate consequence relation $\sqsupseteq$, and the inference relation δ induced by the previous inferences performed on S.

Theorem 2 gives a syntactic criterion to check whether a couple of fixed natural numbers defines a loop. We now show how to test efficiently that (i, j)

Algorithm 1 $\textsc{Cycle}_1(S, i, j)$

```
S_0 ← {n ≢ k, k ∈ [i, i+j[}
if S[i] ⊬_δ S_0 then
    return false
end if
Choose a minimal set S_init ⊆ S[i] s.t. S_init ⊢_δ S_0
S_loop ← ∅
while ∃C ∈ S_init | S_loop ⋢ {C↓_j} do
    if ∃D ∈ S[i+j] | D ⊒ C↓_j then
        Choose D ∈ S[i+j] such that D ⊒ C↓_j
        S_loop ← S_loop ∪ {D}
        if S[i] ⊬_δ D then
            return false
        else
            Choose a set S' ∈ S[i] such that S' ⊢_δ {D}
            S_init ← S_init ∪ S'
        end if
    else
        return false
    end if
end while
return true
```

is a cycle. Two distinct algorithms are presented. The algorithm $\textsc{Cycle}_1$ is the most straightforward and uses a smallest fixpoint algorithm: it starts by considering the minimal possible set S_{init}, namely the set of n-clauses in $S[i]$ that are necessary to derive all the clauses $n \not\equiv k$ for $k \in [i, i+j[$. According to Definition 10, the condition $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$ must hold. Thus, for each clause $\mathcal{C}$ in S_{init}, the algorithm checks whether there exists a n-clause $\mathcal{D}$ in $S[i+j]$ such that $\mathcal{D} \sqsupseteq \mathcal{C}\downarrow_j$. If this is not the case, then (i, j) cannot be a cycle and the algorithm stops. Otherwise, all the ancestors of $\mathcal{D}$ occurring in $S[i]$ must be added to S_{init}, so that the condition $S_{init} \vdash_\delta S_{loop}$ (in Definition 10) holds. The algorithm runs until a fixpoint is reached, in which case a pair of sets of n-clauses (S_{init}, S_{loop}) satisfying the conditions of Definition 10 has been obtained. The drawback of this algorithm is that it is non-deterministic. Indeed, for a given n-clause $\mathcal{C}$, there may exist several n-clauses $\mathcal{D}$ satisfying the desired condition (unless $\sqsupseteq$ is the identity relation). Similarly, the ancestors of $\mathcal{D}$ in $S[i]$ are not unique and can be chosen arbitrarily. To ensure completeness all these branches must be explored, yielding an exponential number of immediate entailment tests (although the number of iteration steps is polynomial w.r.t. the size of S).

The algorithm $\textsc{Cycle}_2$ is slightly more complex, and is based on a greatest fixpoint algorithm. The idea is to start by adding to S_{init} all the n-clauses in $S[i]$. Then S_{loop} is obtained by considering all the n-clauses in $S[i+j]$ that are generated from S_{init}. In order to ensure that the condition $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$ of Definition 10 holds, we remove from S_{init} the n-clauses $\mathcal{C}$ such that there is no n-clause $\mathcal{D}$ in $S[i+j]$ with $\mathcal{D} \sqsupseteq \mathcal{C}\downarrow_j$. If the removed n-clause $\mathcal{C}$ is an ancestor of a

Algorithm 2 $\textsc{Cycle}_2(S, i, j)$

$S_0 \leftarrow \{n \not\simeq k, k \in [i, i+j[\}$
$S_{init} \leftarrow S[i]$
if $S_{init} \not\vdash_\delta S_0$ **then**
  return `false`
end if
$S_{loop} \leftarrow \{\mathcal{D} \in S[i+j] \mid S_{init} \vdash_\delta \{\mathcal{D}\}\}$
while $\exists \mathcal{C} \in S_{init} \mid S_{loop} \not\sqsupseteq \{\mathcal{C}{\downarrow_j}\}$ **do**
  $S_{init} \leftarrow S_{init} \setminus \{\mathcal{C}\}$
  if $S_{init} \not\vdash_\delta S_0$ **then**
    return `false`
  end if
  Remove from S_{loop} all the n-clauses $\mathcal{D}$ s.t. $S_{init} \not\vdash_\delta \{\mathcal{D}\}$
end while
return `true`

clause $n \not\simeq k$ for some $i \in [i, i+j[$ then no cycle possibly exists, and the algorithm stops. Otherwise, the deletion of $\mathcal{C}$ from S_{init} yields the removal of the n-clauses in S_{loop} that are generated from this clause (so that the invariant $S_{init} \vdash_\delta S_{loop}$ holds). This algorithm is deterministic and thus involves a polynomial number of immediate entailment tests (since it is clear that the number of iterations is polynomially bounded by the size of the initial set S). Its actual complexity depends of course on the relation $\sqsupseteq$: it is polynomial if $\sqsupseteq$ can be tested in polynomial time (for instance if $\sqsupseteq$ is the identity or inclusion relations). If $\sqsupseteq$ is the subsumption relation then it is exponential. The main drawback of $\textsc{Cycle}_2$ w.r.t. the previous algorithm $\textsc{Cycle}_1$ is that the handled clause sets are usually larger since the whole set of n-clauses must be considered right from the beginning. Thus the first algorithm may be more adapted to huge clause sets, or if the immediate entailment relation is the identity.

Theorem 3. *The algorithm* $\textsc{Cycle}_1$ *and* $\textsc{Cycle}_2$ *are terminating, sound and complete, i.e.,* $\textsc{Cycle}_1(S, i, j) =$ `true` *iff* $\textsc{Cycle}_2(S, i, j) =$ `true` *iff* (i, j) *is cycle for* S *(w.r.t. the relations* $\sqsupseteq$ *and* δ*)*

Proof. We consider the two algorithms separately.

Algorithm $\textsc{Cycle}_1$**:**

Termination is immediate since at each iteration step, the size of S_{init} increases strictly (and it is bounded by the size of the whole set of n-clauses). If $S[i] \not\models \{n \not\simeq k, k \in [i, i+j[\}$ then by Definition 10, (i, j) cannot be a cycle. Otherwise, according again to Definition 10, the set S_{init} must contain a set of n-clauses entailing $\{n \not\simeq k, k \in [i, i+j[\}$ (w.r.t. $\vdash_\delta$). The end-condition of the while loop ensures that $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$. Furthermore, the invariant $S_{init} \vdash_\delta S_{loop}$ necessarily holds, since each time a clause $\mathcal{D}$ is added into S_{loop}, a set S' such that $S' \vdash_\delta \{\mathcal{D}\}$ is added to S_{init}. Finally, all the n-clauses that are added in S_{init} during the loop are in $S[i]$, thus the invariant $S_{init} \subseteq S[i]$ holds. Consequently, after the while loop, all the conditions of Definition 10 hold, and thus (i, j) must

be a cycle. Conversely, if (i, j) is a cycle, then it is easy to check that a run of Algorithm $\textsc{Cycle}_1$ exists in which `false` is never returned. It suffices to choose the clauses $\mathcal{D}$ and S' in such a way that $\mathcal{D} \in S_{loop}$ and $S' \subseteq S_{init}$ (where S_{loop} and S_{init} correspond to the sets in Definition 10). This is always possible, since by definition we must have $S_{init} \vdash_\delta S_{loop}$ and $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$, with $S_{init} \subseteq S[i]$ and $S_{loop} \subseteq S[i+j]$.

Algorithm $\textsc{Cycle}_2$**:**

Termination is immediate since at each iteration step, the size of S_{init} decreases strictly. Again, if $S[i] \not\models \{n \not\simeq k, k \in [i, i+j[\}$ then by Definition 10, (i, j) cannot be a cycle. Otherwise, we must have $S_{init} \subseteq S[i]$ and $S_{loop} \subseteq S[i+j]$. As for the previous algorithm, the end-condition of the while loop ensures that $S_{loop} \sqsupseteq S_{init}{\downarrow_j}$. Furthermore, the invariant $S_{init} \vdash_\delta S_{loop}$ still holds, by definition of the last instruction in the while-loop. Consequently, after the while loop, all the conditions of Definition 10 hold, and thus (i, j) must be cycle.

Conversely, if (i, j) is a cycle, then the algorithm returns `true`. Indeed, the sets S'_{init} and S'_{loop} corresponding to Definition 10 necessarily occur at each iteration step in the actual sets S_{init} and S_{loop} computed by the algorithm. By definition, no n-clause $\mathcal{C} \in S'_{init}$ can be removed from S_{init}, since we have $S_{loop} \sqsupseteq \{\mathcal{C}{\downarrow_j}\}$. Similarly, no clause $\mathcal{D} \in S'_{loop}$ can be deleted from S_{loop} since we must have $S_{init} \vdash_\delta \{\mathcal{D}\}$. Therefore, the condition $S_{init} \not\vdash_\delta S_0$ can never hold (since $S'_{init} \vdash_\delta S_0$).

Example 3. We consider the following clause set:

$$
\begin{array}{ll}
1 & p(x) \not\simeq \texttt{true} \vee p(succ(x)) \simeq \texttt{true} \\
2 & q(x) \simeq \texttt{true} \vee p(succ(x)) \simeq \texttt{true} \\
3 & f(succ(x)) \simeq f(x) \\
4 & p(0) \simeq \texttt{true} \\
5 & [p(x) \not\simeq \texttt{true} \mid n \simeq x] \\
6 & [f(x) \simeq a \mid n \simeq x] \\
7 & [g(x) \simeq a \mid n \simeq x]
\end{array}
$$

The following clauses are derived:

$$
\begin{array}{lcl}
8 & [\square \mid n \simeq 0] & \text{(superposition, 4, 5)} \\
9 & [p(x) \not\simeq \texttt{true} \mid n \simeq succ(x)] & \text{(superposition, 1,5)} \\
10 & [f(x) \simeq a \mid n \simeq succ(x)] & \text{(superposition, 3,6)} \\
11 & [q(x) \simeq \texttt{true} \mid n \simeq succ(x)] & \text{(superposition, 2,5)}
\end{array}
$$

We illustrate how the two algorithms run on this example. Note that Clauses $1, 4, 5$ are sufficient for unsatisfiability (Clause 5 asserts that $\neg p(n)$ holds for some natural number n, which is impossible since Clauses 1 and 4 entail that $p(succ^k(0))$ holds for every $k \in \mathbb{N}$), the other clauses are added only to illustrate how the algorithms work and to emphasize the differences between them. We take $i = 0$, $j = 1$ and we use the identity as the immediate consequence relation $\sqsupseteq$. We have $S_0 = \{8\}$, $S[i] = \{5, 6, 7\}$, $S[i+j] = \{9, 10, 11\}$, $S[\bot] = \{1, 2, 3, 4\}$.

Algorithm $\textsc{Cycle}_1$**:** The algorithm first chooses a set of clauses $S_{init} \subseteq S[i]$ such that $S_{init} \vdash_\delta S_0$. The parents of Clause 8 are $4 \in S[\bot]$ and $5 \in S[i]$, thus

according to Definition 8 we have $\{5\} \vdash_\delta \{8\}$. Therefore, S_{init} is set to $\{5\}$. Then S_{loop} is initialized to $\emptyset$. Clause 5 occurs in S_{init} and we have $5{\downarrow_1} \notin S_{loop}$ (since $S_{loop} = \emptyset$). Thus the algorithm chooses a clause $\mathcal{D} \in S[i+j]$ such that $\mathcal{D} \sqsupseteq 5{\downarrow_1}$. Clause 5 is $[p(x) \not\simeq \texttt{true} \mid n \simeq x]$, thus $5{\downarrow_1}$ is $[p(x) \not\simeq \texttt{true} \mid n \simeq succ(x)]$, hence $5{\downarrow_1} = 9$. Therefore, the only solution is $\mathcal{D} = 9$. This clause is added to S_{loop}. The algorithm checks that $S[i] \vdash_\delta \mathcal{D}$ and adds to S_{init} a minimal set of clauses S' such that $S' \vdash_\delta \{\mathcal{D}\}$. Clause 9 is deduced from Clause 1, which occurs in $S[\bot]$, and Clause 5, which occurs in $S[i]$, thus the only solution is $S' = \{5\}$. Therefore S_{init} is not affected (since it already contains 5). The while-loop ends because the only clause in $S_{init}{\downarrow_1}$ occurs in S_{loop}. The pair $(0, 1)$ is a cycle, corresponding to the sets $S_{init} = \{5\}$ and $S_{loop} = \{9\}$.

Algorithm $\textsc{Cycle}_2$**:** S_{init} is initialized with $S[i]$, i.e. $\{5, 6, 7\}$. Then the algorithm checks that $S_{init} \vdash_\delta S_0$, and initializes S_{loop} with the set of clauses $\mathcal{D} \in S[i+j]$ such that $S_{init} \vdash_\delta \mathcal{D}$. All the clauses in $S[i+j]$ are obtained from clauses in S_{init} and $S[\bot]$, thus we have $S_{loop} = \{9, 10, 11\}$. Then the algorithm checks whether S_{init} contains a clause $\mathcal{C}$ such that $\mathcal{C}{\downarrow_j} \notin S_{loop}$. The only clause satisfying this condition is Clause 7. Thus this clause is removed from S_{init}, yielding $\{5, 6\}$ and the clauses $\mathcal{D}$ such that $\{5, 6\} \not\vdash_\delta \mathcal{D}$ are removed from S_{loop}. Here all the clauses in S_{loop} can be deduced from $\{5, 6\}$ and clauses in $S[\bot]$ thus S_{loop} is not affected. Then the algorithm stops and returns `true`. The obtained sets are $S_{init} = \{5, 6\}$ and $S_{loop} = \{9, 10\}$. Note that, compared to the previous case, the sets contain an additional clause (namely 6/10), which occurs in the generated inductive invariant, but actually plays no role in the proof (these clauses can be identified and eliminated afterwards by applying reachability analysis algorithms on the inference graph).

6 Implementation

Our calculus has been implemented as a research prototype, using the system Prover9 [17] as an inference engine. While any other superposition-based prover could be used instead, the system is not used as a mere "black box": the procedures and data-structures had to be adapted in order to handle the specific features of the calculus: arithmetic constraints, normalization of clauses, etc. The program uses the usual "given clause algorithm" of Prover9, and calls Algorithm $\textsc{Cycle}_2$ to check whether a given pair (i, j) is a cycle, using the subsumption relation as an immediate consequence relation. The test is triggered at each iteration of the main loop, and only if all the clauses $n \not\simeq k$ for $k \in [0, i+j[$ have been generated (thus a cycle is detected only if this leads to an immediate stop). A refutation is obtained if the system generates a set of n-clauses of the form $\{\Box\}$ or $\{n \not\simeq 0, \ldots, n \not\simeq k-1, n < k\}$, which is obviously unsatisfiable. The last clause $n < k$ is usually derived by cycle detection (with $k = i+j$), but it can also be derived by the superposition calculus alone, in simple cases in which the theorem can be proven without induction. If the system is fed with an unsatisfiable set of standard clauses then the empty clause $\Box$ can be generated as usual. Our proof procedure is not complete in general (the logic is not semi-decidable). We

use heuristics to preserve the partial completeness results in [16]. For instance a greater weight is associated with the symbol *succ* to ensure that the literals containing a maximal arithmetic expression are selected with the highest priority, and some inferences are blocked to ensure that $S[i] \models S[i+1]$.

Now, let's prove the theorem:

$$\forall n \in \mathbb{N}\ \forall a_1, \ldots, a_n \quad a_1 \times a_2 \times \cdots \times a_n = a_n \times a_{n-1} \times \cdots \times a_1 \qquad (1)$$

We show the corresponding input file:

```
formulas(sos).
N(x) | p(x) != q(x).
p(0)=1.
q(0)=1.
p(s(x)) = p(x)*a(x).
q(s(x)) = a(x)*q(x).
*(x,1)=x.
*(1,x)=x.
x*y= u*v | x!=u | y != v.
x*y = y*x.
end_of_list.
```

Our tool has almost the same input format than Prover9, we just have to add the constraints to the clauses, a constraint of the form $n \not\simeq t$ (where t is a term of sort `nat`) is written $N(t)$ and attached to the clause as a literal. The first clause of the input file corresponds to $[p(x) \not\simeq q(x) \mid n \simeq x]$, which is also the negation of (1), p(x) and q(x) encode the terms $a_1 \times \ldots \times a_n$ and $a_n \times \ldots \times a_1$ respectively. We show the output file generated by our tool:

```
============================== PROOF =================================
% Proof at 0.02 seconds.
% Given clauses 17.
S_init  :
(67: N(v0) | -=(0,1) | -=(1,v0) .
   2: N(v0) | -=(q(v0),p(v0)).)
S_loop :
(107: N(s(v0)) | -=(0,1) | -=(1,v0) .
   85: N(s(v0)) | -=(q(v0),p(v0)).)
The empty clauses  :
 (12: N(0).)
============================== end of proof ==========================
```

The output file contains the running time, the number of given clauses, the two clause sets S_{init}, S_{loop} and finally the pure constraint clause $N(0)$ which corresponds to the clause $[\square \mid n \simeq 0]$. As in Example 3, the obtained inductive invariant contains an additional clause that plays no role in the derivation.

7 Experimentation

In this section we provide some examples of application of our work. All the presented problems require induction and thus are out of the scope of first-order theorem provers. We first present some examples in propositional logic. We consider an n-bit sequential adder circuit i.e. a circuit which computes the sum of two bit-vectors of length n. Such a circuit is built by composing n 1-bit adders. The ith bits of each operand are written p_i and q_i. r_i is the ith bit of the result and c_{i+1} is carried over to the next bit (thus $c_1 = 0$). We set the notations ($\oplus$ denotes exclusive or): $\mathit{Sum}_i(p, q, c, r) \stackrel{\text{def}}{=} r_i \Leftrightarrow (p_i \oplus q_i) \oplus c_i$ and $\mathit{Carry}_i(p, q, c) \stackrel{\text{def}}{=} c_{i+1} \Leftrightarrow (p_i \wedge q_i) \vee (c_i \wedge p_i) \vee (c_i \wedge q_i)$. Then the formula: $\mathit{Adder}(p, q, c, r) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} \mathit{Sum}_i(p, q, c, r) \wedge \bigwedge_{i=1}^{n} \mathit{Carry}_i(p, q, c) \wedge \neg c_1$ with the constraint $n \geq 1$, schematises the adder circuit (it states that r encodes the sum of p and q). In order to test the satisfiability of such schemata of propositional formulæ, we have implemented an algorithm transforming automatically (in polynomial time) any propositional schema built on iterated connectives of the form $\bigwedge_{i=a}^{n+b} \phi$ or $\bigvee_{i=a}^{n+b} \phi$ (such as the ones modeling the Adder circuit) into a sat-equivalent set of n-clauses. This algorithm works by introducing a monadic predicate (of domain `nat`) for every iteration occurring in the initial formula, and by adding axioms to specify the interpretation of these predicates by induction on the natural numbers. For instance, the schema $\bigvee_{i=0}^{n} p_i$ can be denoted by the atom $[q(x) \simeq \texttt{true} \mid n \simeq x]$, with the axioms: $\{q(0) \not\simeq \texttt{true} \vee p(0) \simeq \texttt{true}, q(\mathit{succ}(x)) \not\simeq \texttt{true} \vee p(\mathit{succ}(x)) \simeq \texttt{true} \vee q(x) \simeq \texttt{true}\}$ (the formal description of the transformation algorithm is omitted due to space restrictions, it can be found in [2]). Several properties of the *Adder* can then be automatically checked, such as commutativity or associativity. We have encoded two different versions of the Adder (the carry propagate and ripple-carry adders respectively) and proved some elementary properties of these circuits.

We have also considered examples coming from an interesting application of schemata languages in proof theory, developed in the context of the ASAP project (see `http://membres-lig.imag.fr/peltier/ASAP/`). The method CERES (see for instance [3]) is an algorithm for cut-elimination in first-order logic that is more efficient that the standard (reductive) approach. It works by extracting from the considered (non-analytic) proof π an unsatisfiable set of clauses $S(\pi)$, called the *characteristic set* of π, which is defined in such a way that any resolution proof of $S(\pi)$ can be automatically transformed into an analytic proof of S. It has been extended to schemata of first-order proofs in [9, 19, 10], in order to handle mathematical proofs using induction (which cannot be expressed in first-order logic and which, as well-known, do not admit cut elimination algorithms). The obtained characteristic set is then not a set of clauses in the usual sense, but rather a schema of clause sets, which can be expressed as a set of n-clauses, and handled using our calculus. We provide the running times for the characteristic sets obtained from simple proofs (the formal definition of the schemata is omitted for conciseness, the purely propositional ones can be found in the RegSTAB webpage at `http://regstab.forge.ocamlcore.org/`

and the first-order one can be found in [19, 10]). Finally we consider some simple inductive properties, for instance we prove that if we perform an arbitrary number of permutations on a sequence containing an element a then the final sequence still contains a.

The obtained results are depicted below. We provide for each example, the running time, the number of calls to $\textsc{Cycle}_2$ and the number of generated clauses.

Example	Time (s)	# of calls to $\textsc{Cycle}_2$	# generated clauses
Ripple-carry adder ($A + 0 = A$)	0.48	336	33833
Ripple-carry adder (commutativity)	0.03	102	2003
Ripple-carry adder (associativity)	0.09	207	10154
Ripple-carry adder ($3 + 4 = 7$)	0.06	71	9989
Unicity of the result (ripple-carry)	0.7	150	50901
Carry-propagate adder (commutativity)	0.02	14	1980
Carry-propagate adder (associativity)	0.01	20	3972
Equivalence between the ripple-carry and the carry-propagate adders	0.03	14	1980
CERES ex1 (Propositional)	0.01	40	995
CERES ex2 (Propositional)	0.03	216	4106
CERES (First order)	0.01	23	49
Totality of $<$ ($n_1 \geq n_2 \vee n_1 < n_2$)	0.01	47	185
$\bigwedge_{i=1}^{n} p_i > 0 \Rightarrow p_1 \times \cdots \times p_n > 0$	0.01	8	59
Permutation (triplet)	0.01	17	280

The results show that the cycle detection algorithm is efficient, even for sets containing thousands of clauses.

8 Conclusion

We have presented a method to enrich superposition-based theorem proving with inductive reasoning capabilities. To this purpose, we have devised algorithms to detect cycles in the superposition derivation in an automatic way. These cycles correspond to inductive invariants and allow one to prune infinite superposition derivations. Our method has been implemented and some examples of application have been presented. Future work includes the extension of the implementation, for instance by devising refined criteria for triggering the application of the cycle detection procedure or by introducing new techniques for performing this detection in an incremental way.

References

1. E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic sup(la). In S. Ghilardi and R. Sebastiani, editors, *FroCoS 2009*, volume 5749 of *LNCS*, pages 84–99. Springer, 2009.

2. V. Aravantinos, M. Echenim, and N. Peltier. A resolution calculus for first-order schemata. *Fundamenta Informaticae*, 2013. Accepted for publication, to appear.
3. M. Baaz and A. Leitsch. Towards a clausal analysis of cut-elimination. *J. Symb. Comput.*, 41(3-4):381–410, 2006.
4. L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierachic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.
5. G. Barthe and S. Stratulat. Validation of the javacard platform with implicit induction techniques. In *RTA 2003*, volume 2706 of *LNCS*, pages 337–351. Springer, 2003.
6. P. Baumgartner and C. Tinelli. Model Evolution with Equality Modulo Built-in Theories. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 85–100. Springer, 2011.
7. A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE, an automatic theorem prover. In *Proceedings of LPAR'92*, volume 624, pages 460–462. Springer-Verlag, 1992.
8. H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 14, pages 913–962. North-Holland, 2001.
9. T. Dunchev. *Automation of cut-elimination in proof schemata*. PhD thesis, T.U. Vienna, 2012.
10. T. Dunchev, A. Leitsch, M. Rukhaia, and D. Weller. Ceres for first-order schemata, 2013. Research Report, `http://arxiv.org/abs/1303.4257`.
11. S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning*, volume 7364 of *LNCS*, pages 241–255. Springer Berlin Heidelberg, 2012.
12. Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *CAV 2009*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
13. J. Giesl and D. Kapur. Decidable classes of inductive theorems. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *LNCS*, pages 469–484. Springer, 2001.
14. J. Giesl and D. Kapur. Deciding inductive validity of equations. In F. Baader, editor, *CADE*, volume 2741 of *LNCS*, pages 17–31. Springer, 2003.
15. M. Horbach and C. Weidenbach. Superposition for fixed domains. *ACM Trans. Comput. Logic*, 11(4):1–35, 2010.
16. A. Kersani and N. Peltier. Completeness and Decidability Results for First-order Clauses with Indices. In *Proceedings of CADE'13 (24th International Conference on Automated Deduction)*. Springer, 2013.
17. W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.
18. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.
19. M. Rukhaia. *CERES in Proof Schemata*. PhD thesis, T.U. Vienna, 2012.
20. S. Stratulat. Automatic 'descente infinie' induction reasoning. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2005)*, volume 3702 of *LNCS*, pages 262–276. Springer, 2005.