Completeness and Decidability Results for First-order Clauses with Indices

Abdelkader Kersani, Nicolas Peltier

Completeness and Decidability Results for First-order Clauses with Indices*

Abdelkader Kersani and Nicolas Peltier

University of Grenoble (LIG, CNRS)

Abstract. We define a proof procedure that allows for a limited form of inductive reasoning. The first argument of a function symbol is allowed to belong to an inductive type. We will call such an argument an *index*. We enhance the standard superposition calculus with a loop detection rule, in order to encode a particular form of mathematical induction. The satisfiability problem is not semi-decidable, but some classes of clause sets are identified for which the proposed procedure is complete and/or terminating.

1 Introduction

We consider first-order clauses in which some of the function or predicate symbols are indexed by a particular kind of terms. The difference between these indices and the usual arguments is that the former are interpreted as ground terms constructed on a given signature (i.e. on an inductively defined domain), whereas the latter are interpreted arbitrarily (in the usual way). Consider the following example:

$$p_0(a) \wedge (\forall i\ \forall x\ p_i(x) \Rightarrow p_{s(i)}(f(x))) \wedge \forall x\ \neg p_n(x)$$

This formula is unsatisfiable if the constant n is interpreted as a term constructed on the signature $\{0, s\}$, i.e. as an element of $\mathbb{N}$. Indeed, for any $m \in \mathbb{N}$, the formula $p_m(f^m(a))$ can be derived from the first two conjuncts, yielding a contradiction with the third conjunct. However, if the indices are interpreted as ordinary terms, then the formula is obviously satisfiable. If the value of n is fixed (i.e. $n = 1, 2, \ldots$) then any first-order prover can easily establish the unsatisfiability of the formula. However, proving that it is unsatisfiable for every $n \in \mathbb{N}$ is a much harder problem, which obviously requires the use of mathematical induction. The previous formula can be viewed as a *schema* of clause sets, in the sense that, to transform this set into a standard clause set, one has to replace the "parameter" n by a ground term $s^m(0)$. Schemata of formulæ arise naturally in many applications of Automated Theorem Proving, in particular for formalizing parameterized systems (for instance circuits depending on the number of bits or layers [14]), for the modelling of dynamic systems (where the indices encode the time: $p_i(\boldsymbol{t})$ holds iff $p(\boldsymbol{t})$ is true at instant i), or for the formalization of inductive proofs in mathematics (the index then represents the induction parameter, see

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

[5] for an example of use of this technique). In this paper, we devise a proof procedure for testing the satisfiability of such formulæ. The proposed inference system uses the usual rules of the superposition calculus (with a specific formalism in which parameters are abstracted away from the clauses), together with a new rule which encodes a form of mathematical induction. Due to well-known theoretical limitations, the satisfiability problem is not semi-decidable in general (see Proposition 1), but we devise some additional criteria that ensure completeness or termination. The indices are not necessarily natural numbers: they can be interpreted as any ground term on a given signature, provided all the (non-constant) symbols are monadic (i.e. the indices are interpreted as words).

The rest of the paper is structured as follows. In Section 2 we define the syntax and semantics of clausal logic with indices. In Section 3, we adapt the usual superposition calculus. In Section 4 we define a loop detection rule that strictly increases the power of the superposition calculus and we show examples of application. In Section 5 and 6, some abstract conditions ensuring refutational completeness and/or termination are devised. In Section 7 we provide an example of a syntactic class of clause sets fulfilling the previous conditions and Section 8 concludes the paper. Due to space restrictions, some proofs are omitted. Missing proofs can be found in [18].

Related work

Our approach is strongly related to the "superposition calculus for fixed domain" procedure defined in [16]. Actually, the "superposition part" of our calculus is essentially equivalent to that in [16] and also to that in [7], which is designed to handle "hybrid" reasoning, i.e. reasoning combining the use of a theory-specific procedure with the superposition calculus for handling the generic part of the proof. However, in our approach the use of the fixed domain terms is more restricted: they only appear as distinguished arguments in the terms and *not* as ordinary arguments. Furthermore, we only consider formulæ with a unique parameter. This distinction between indices and ordinary terms reduces the scope of the method but permits to obtain much stronger completeness and decidability results. The proof procedure in [16] is not complete in general, since, for refuting the considered formula, an *infinite* set of empty constrained clauses must be generated in some cases. Some completeness and decidability results are presented in [15], however they are based on many additional conditions which do not hold in our case: all the clauses must be Horn, all the symbols must be monadic etc. The loop detection rule proposed in the present paper is also very different from the inductive rules defined in [16] and [15]. Therefore, the two approaches can be viewed as complementary (a more detailed comparison is provided in Section 4). Our work is also strongly related to inductive theorem proving. Explicit induction approaches (see for instance [10] or [8]) are often used by proof assistants, and powerful heuristics are employed to derive automatically the appropriate induction schemes [11]. Implicit induction schemes are used in rewrite-based theorem provers [9], whereas inductionless induction (see for example [17, 12]) uses proof by consistency to reduce the inductive validity to a mere satisfiability check. Very few completeness or termination results exist for

such provers and our language does not fall in the scope of the known complete classes. In general, inductive theorem proving requires strong human guidance, especially for specifying the needed inductive lemmata. In contrast, our procedure, although more focused in this scope, is *purely automatic*: the loop detection rule allows one to generate automatically inductive invariants. Both the implicit and the inductionless induction approaches handle universal properties: the considered goals are of the form $\phi \models_{ind} \forall \boldsymbol{x} \psi$, where $\models_{ind}$ denotes the inductive logical consequence relation, ϕ is the axiomatization and ψ is a quantifier-free formula (usually a clause). Our work departs from these approaches because the goals we consider are rather of the form $\phi \models \forall n Q_1 x_1 \ldots Q_n x_n \psi$ (where n denotes the parameter, $x_1, \ldots, x_n$ are standard variables and $Q_1, \ldots, Q_n$ are quantifiers). Indeed the value of the standard first-order variables can possibly depend on the value of the parameter. Our calculus is also related to the proof procedure described in [1] which handles schemata of propositional formulæ indexed by integers. The scope of the present paper is however much larger, both for the base language (first-order logic instead of propositional logic) and for the type of the indices (terms – or words – instead of integers). Our calculus is also strictly more powerful than the one presented in [3], which only handles first-order clauses without equality. Besides, the completeness results in [3] only hold for purely propositional schemata.

2 Preliminaries

In this section, we define the syntax and semantics of the considered logic. We assume the reader is familiar with the usual notions in logic and automated deduction [22]. We consider first-order terms, built as usual on a sorted signature Σ and on a set of variables X, in which some of the (function or constant) symbols are indexed by terms of some special sorts. The sorts are used mainly to distinguish the indices from the ordinary terms. The set of sort symbols is thus partitioned into two disjoint sets $\mathcal{S}_I$ and $\mathcal{S}_T$, where $\mathcal{S}_I$ denotes the sorts of the indices and $\mathcal{S}_T$ the sorts of the ordinary terms. We assume that the profile of every non-constant symbol f is either of the form $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{s}$, where $\mathtt{s}_2, \ldots, \mathtt{s}_n, \mathtt{s} \in \mathcal{S}_T$, and $\mathtt{s}_1 \in \mathcal{S}_I \cup \mathcal{S}_T$ or of the form $\mathtt{s} \to \mathtt{s}'$, where $\mathtt{s}, \mathtt{s}' \in \mathcal{S}_I$, i.e. all function symbols of a sort in $\mathcal{S}_T$ have at most one argument of a type in S_I and all (non-constant) function symbols of a sort in $\mathcal{S}_I$ are monadic and have a domain in $\mathcal{S}_I$. A *predicate symbol* is a function symbol of profile $\mathtt{s} \to \mathtt{bool}$. A variable of a sort in $\mathcal{S}_I$ is an *index variable*.

For readability, a term of head symbol $f : \mathtt{s}_1, \ldots, \mathtt{s}_n \to \mathtt{s}$ such that $\mathtt{s}_1 \in \mathcal{S}_I$ and $\mathtt{s}_2, \ldots, \mathtt{s}_n \in \mathcal{S}_T$ and of arguments $i, t_1, \ldots, t_n$ will be written $f_i(t_1, \ldots, t_n)$ (i is called an *index term*). The other terms are written as usual. This convention allows one to clearly distinguish the induction terms from the standard ones. If t is a term and p is a position (i.e. a finite sequence of natural numbers) then $t|_p$ denotes the subterm of t at position p (defined as usual). If v is a term then $t[v]_p$ denotes the term obtained from t by replacing the subterm at position p by v.

An *atom* is of the form $t_1 \simeq t_2$, where t_1, t_2 are terms of the same sort in $\mathcal{S}_T$ (equations between index terms are forbidden). A *literal* is either an atom

(*positive* literal) or the negation of an atom (*negative* literal). A *clause* is a finite multiset of literals (written as a disjunction) and $\Box$ denotes the empty clause. Let α be a special constant symbol (not occurring in Σ) of a sort in $\mathcal{S}_I$. Throughout this paper, $\mathfrak{C}$ denotes some particular class of clauses.

Definition 1. *Let α be a constant symbol of a sort in S_I. An* α-clause *is an expression of the form $(\alpha \not\approx i_1) \vee \ldots \vee (\alpha \not\approx i_n) \vee C$ (with possibly $n = 0$) where:*

- *C is a clause.*
- *$i_1 \ldots i_n$ are terms of the same sort as α.*
- *Either C is empty, or all the variables in $i_1, \ldots, i_n$ occur in C.*

If, moreover, the clause C belongs to the class of clauses $\mathfrak{C}$, then $(\alpha \not\approx i_1) \vee \ldots \vee (\alpha \not\approx i_n) \vee C$ is an $(\alpha, \mathfrak{C})$-clause. *We call the α-clause* normalized *if $n \in \{0, 1\}$.*

The constant α is called the *parameter*. The class $\mathfrak{C}$ is mainly useful to restrict the syntactic form of the considered expressions. It is assumed to be fixed once and for all in the rest of the paper. Notice that, to avoid confusion, we use the symbol $\approx$ instead of $\simeq$ to denote equations between indices (i.e. in which α is involved). Notice also that the symbol α cannot occur in a term or literal (since $\alpha \notin \Sigma$). Therefore, a property such as $a_\alpha \simeq b$ for instance is to be written as $\alpha \not\approx x \vee a_x \simeq b$, where x is a variable. This idea of abstracting away terms and replacing them by variables is commonly used in the superposition framework to delay reasoning on some particular terms (see for instance [7]).

For every expression (term, atom, literal or clause) e, $\mathrm{var}(e)$ denotes the set of variables occurring in e. If $\mathrm{var}(e) = \emptyset$ then e is *ground*. A *substitution* σ is a function mapping every variable x to a term $x\sigma$ of the same sort as x. The *domain* $dom(\sigma)$ of σ is the set of variables x such that $x\sigma \neq x$. For every expression e, $e\sigma$ denotes the expression obtained from e by replacing every variable x by $x\sigma$. A substitution σ is *ground* iff for every $x \in dom(\sigma)$, $x\sigma$ is ground. A *renaming* is an injective substitution σ such that $x\sigma \in X$ for every $x \in dom(\sigma)$. The notions of *unifiers* and *most general unifiers* (*mgu*) are defined as usual. If t and s are two terms, we write $t \succeq s$ if $s = t\sigma$, for some substitution σ. We write $t \succ s$ if $t \succeq s$ and $s \not\succeq t$. A set of terms T of the same sort $\mathtt{s}$ is *covering for a term t* of sort $\mathtt{s}$ iff for every ground substitution θ, there exists $s \in T$ such that $t\theta \preceq s$. It is *covering* if it is covering for all terms of sort $\mathtt{s}$. The problem of testing whether a given set of terms is covering or not for a term t is decidable [13].

Definition 2. *An* interpretation $\mathcal{I}$ *is a pair $(=_\mathcal{I}, \mathcal{I}(\alpha))$ where $=_\mathcal{I}$ is a congruence on the ground terms whose sort is in $\mathcal{S}_T$ and $\mathcal{I}(\alpha)$ is a ground term of the same sort as α. An interpretation $\mathcal{I}$* validates*:*

- *A ground literal $t_1 \simeq t_2$ (resp. $t_1 \not\simeq t_2$) iff $t_1 =_\mathcal{I} t_2$ (resp. $t_1 \neq_\mathcal{I} t_2$).*
- *A ground clause $C \in \mathfrak{C}$ iff it validates at least one literal in C.*
- *A ground $(\alpha, \mathfrak{C})$-clause $\alpha \not\approx i_1 \cdots \vee \alpha \not\approx i_n \vee C$ iff either $\mathcal{I}(\alpha) \neq i_j$ for some $j \in [1, n]$ or $\mathcal{I}$ validates C.*
- *A non-ground $(\alpha, \mathfrak{C})$-clause C iff for every ground substitution σ of domain $var(C)$, $\mathcal{I}$ validates $C\sigma$.*

– *A set of* $(\alpha, \mathfrak{C})$*-clauses* S *iff it validates every* $(\alpha, \mathfrak{C})$*-clause in* S.

We write $\mathcal{I} \models S$ if $\mathcal{I}$ validates S ($\mathcal{I}$ is a *model* of S) and $S \models S'$ if every model of S is a model of S'.

The logic is obviously not decidable since it is clear that it encompasses first-order logic: if the clauses contain no occurrences of the special constant symbol α, then the value of the parameter is irrelevant, and an interpretation is simply a congruence on the set of Herbrand terms. In this case our semantics coincides with the usual one. The following theorem states that it is not even semi-decidable.

Theorem 1. *The unsatisfiability problem is not semi-decidable for* $(\alpha, \mathfrak{C})$*-clauses, if* $\mathfrak{C}$ *is the whole class of first-order clauses with one index variable.*

Proof. The proof is by reduction to the Post correspondence problem. Let $u^1, \ldots, u^n$ and $v^1, \ldots, v^n$ be two sequences of words. We construct a set of $(\alpha, \mathfrak{C})$-clauses S such that S is satisfiable iff there exists a sequence of indices $i_1, \ldots, i_m$ such that $u^{i_1} \ldots . u^{i_m} = v^{i_1} \ldots . v^{i_m}$. We use a constant symbol l encoding the sequence $i_1, \ldots, i_m$, with two function symbols *head* and *tail* returning respectively the head and the tail of a list. Words are encoded as usual, as lists of characters. The constant ϵ denotes the empty word and *concat* is a function concatenating two words (its definition is straightforward and is omitted). If $x \in \{u, v\}$ and y denotes a sequence of indices $i_1, \ldots, i_m$, then $sol(y, x)$ denotes the word $x^{i_1} \ldots . x^{i_m}$. The terms $word(u, i)$ and $word(v, i)$ denote the words u^i and v^i respectively We use the following axioms:

– $\neg empty(x) \vee head(x) \simeq 1 \vee \ldots \vee head(x) \simeq n$ (if a sequence is not empty then its head is in $[1, n]$)
– $word(u, i) \simeq u^i$, for every $i \in [1, n]$ (definition of u).
– $word(v, i) \simeq v^i$, for every $i \in [1, n]$ (definition of v).
– $\neg empty(y) \vee sol(y, x) \simeq \epsilon$. If the sequence $i_1, \ldots, i_m$ is empty then the solution $x^{i_1} \ldots . x^{i_m}$ is also empty.
– $empty(y) \vee sol(y, x) \simeq concat(word(x, head(y)), sol(tail(y), x))$. Otherwise, it corresponds to the concatenation of the word x^{i_1} and the solution corresponding to the sequence $i_2, \ldots, i_m$.
– $word(u, l) \simeq word(v, l)$. The two sequences are equal.

We now define a predicate $length_i(l)$ to encode the fact that l has length i.

– $length_0(x) \vee \neg empty(x)$
– $(\neg length_{i+1}(x) \vee \neg empty(x)) \wedge (\neg length_0(x) \vee empty(x))$
– $(length_{i+1}(x) \vee \neg length_i(tail(x))) \wedge (\neg length_{i+1}(x) \vee length_i(tail(x)))$

Finally, the following clauses state that the constant l denotes a finite list of length $\alpha \neq 0$ (notice that this property is the only one that cannot be expressed in first-order logic: otherwise l could denote an infinite or cyclic list): $\alpha \not\approx x \vee length_x(l) \wedge \neg empty(l)$

It is straightforward to check that the previous set of clauses is satisfiable iff there exists a sequence of indices $i_1, \ldots, i_m$ satisfying the desired property.

Proposition 1. *Let $D : \alpha \not\approx i_1 \vee \ldots \vee \alpha \not\approx i_n \vee C$ be an $(\alpha, \mathfrak{C})$-clause, with $n \geq 1$. If $i_1, \ldots, i_n$ are not unifiable then D is a tautology. Otherwise, D is equivalent to $\alpha \not\approx i_1\sigma \vee C$ where σ is an mgu of $i_1, \ldots, i_n$. In this case, $\alpha \not\approx i_1\sigma \vee C$ is the* normalized form *of D.*

From now on, we assume that every $(\alpha, \mathfrak{C})$-clause is normalized. Indeed, by Proposition 1, a clause $\alpha \not\approx i_1 \vee \ldots \alpha \not\approx i_n \vee C$ can be either deleted or replaced by its normalized form.

Superposition calculus:

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

Paramodulation $C \vee t \not\approx s, D \vee u \simeq v \rightarrow (C \vee D \vee t[v]_p \not\approx s)\sigma$
if $\sigma = \mathrm{mgu}(u, t|_p)$, $u\sigma \not\leq v\sigma, t\sigma \not\leq s\sigma$, $t|_p$ is not a variable,
$(t \not\approx s)\sigma \in \mathrm{sel}([C \vee t \not\approx s]\sigma)$, $(u \simeq v)\sigma \in \mathrm{sel}([D \vee u \simeq v]\sigma)$.

Reflection $C \vee t \not\approx s \rightarrow C\sigma$
if $\sigma = \mathrm{mgu}(t, s)$, $(t \not\approx s)\sigma \in \mathrm{sel}([C \vee t \not\approx s]\sigma)$.

Eq. Factorisation $C \vee t \simeq s \vee u \simeq v \rightarrow (C \vee s \not\approx v \vee t \simeq s)\sigma$
if $\sigma = \mathrm{mgu}(t, u)$, $t\sigma \not\leq s\sigma$, $u\sigma \not\leq v\sigma$, $(t \simeq s)\sigma \in \mathrm{sel}([C \vee t \simeq s \vee u \simeq v]\sigma)$.

Fig. 1: The superposition calculus

3 A Superposition Calculus for Indexed Clauses

Our proof procedure is a conservative extension of the superposition calculus [6, 20]. All the rules are applied without any modification, except that disequations containing α are simply ignored (no inference can be applied from or into such disequations).

Let $<$ denote a simplification ordering that is total on ground terms [20]. The ordering $<$ is extended to atoms, literals and clauses using the multiset extension and to $(\alpha, \mathfrak{C})$-clauses simply by ignoring disequations containing α. A literal L is *maximal* in a clause $C \in \mathfrak{C}$ if for every $L' \in C$, $L \not< L'$. We consider a *selection function* sel which maps every clause C to a set of *selected literals* in C. For completeness, we assume that for every clause C, $\mathrm{sel}(C)$ contains either a negative literal or all maximal literals. This selection function is extended to $(\alpha, \mathfrak{C})$-clauses as follows: for every $(\alpha, \mathfrak{C})$-clause $\alpha \not\approx i_1 \vee \ldots \vee \alpha \not\approx i_n \vee C$, $\mathrm{sel}(\alpha \not\approx i_1 \vee \ldots \vee \alpha \not\approx i_n \vee C) \stackrel{\mathrm{def}}{=} \mathrm{sel}(C)$.

We consider the calculus (parameterized by $<$ and sel) of Figure 1. If S is a set of $(\alpha, \mathfrak{C})$-clauses, we write $S \vdash C$ if C is an $(\alpha, \mathfrak{C})$-clause and if there exists a non-tautological $(\alpha, \mathfrak{C})$-clause C' that can be deduced from S by applying one of the rules of Figure 1 such that C is the normalized form of C'.

We also adapt the usual redundancy criteria. A *tautology* is an $(\alpha, \mathfrak{C})$-clause containing two complementary literals, or a literal of the form $t \simeq t$, or a disjunction $\bigvee_{i=1}^{n} \alpha \not\approx t_i$, where $t_1, \ldots, t_n$ are not unifiable. An $(\alpha, \mathfrak{C})$-clause C is *subsumed* by an $(\alpha, \mathfrak{C})$-clause D if there exists a substitution σ such that $D\sigma \subseteq C$. A ground $(\alpha, \mathfrak{C})$-clause C is *redundant* in S if there exists a set of $(\alpha, \mathfrak{C})$-clauses S' such that $S' \models C$, and for every $D \in S'$, D is an instance of an $(\alpha, \mathfrak{C})$-clause in S such that $D \leq C$. A non ground $(\alpha, \mathfrak{C})$-clause C is *redundant* if all its instances are redundant. In particular, every subsumed $(\alpha, \mathfrak{C})$-clause and every tautological clause is redundant. A set of $(\alpha, \mathfrak{C})$-clauses S is *saturated* if every $(\alpha, \mathfrak{C})$-clause C such that $S \vdash C$ is redundant in S.

4 Loop Detection

The superposition calculus is not powerful enough to derive a contradiction from unsatisfiable sets of $(\alpha, \mathfrak{C})$-clauses, even in trivial cases, because it does not take into account the inductive structure of the domain of α. This is well illustrated by the following example.

Example 1. The first example in the Introduction can be encoded as the following set of $(\alpha, \mathfrak{C})$-clauses (where $p_i(t)$ denotes as usual the equation $p_i(t) \simeq \mathit{true}$).

$$1.\ p_0(a) \quad 2.\ \neg p_x(y) \vee p_{s(x)}(f(y)) \quad 3.\ \alpha \not\approx x \vee \neg p_x(y)$$

It is easy to check that the following clauses can be generated by superposition:

4. $\alpha \not\approx 0$	(3,1)	7. $\alpha \not\approx s(s(x)) \vee \neg p_x(y)$	(5,2)
5. $\alpha \not\approx s(x) \vee \neg p_x(y)$	(3,2)	8. $\alpha \not\approx s(s(0))$	(7,1)
6. $\alpha \not\approx s(0)$	(5,1)	...	

It is clear that an infinite set of clauses of the form $\alpha \not\approx s^n(0)$ (for $n \in \mathbb{N}$) can be generated. Since α must be interpreted by a term of the form $s^n(0)$, the set is unsatisfiable (the set $\{s^n(0) \mid n \in \mathbb{N}\}$ is covering). However, no contradiction can be derived in finite time, which shows that the calculus is not complete (note that the logic is not compact).

In this section, we show how to overcome this problem in some cases (by Theorem 1 no general solution is possible). Notice that, in some cases, the calculus can generate a *finite* set of disequations $\{\alpha \not\approx t_1, \ldots, \alpha \not\approx t_n\}$ such that $\{t_1, \ldots, t_n\}$ is covering, in which case one may conclude that the clause set is unsatisfiable (of course it may also directly generate $\square$, if α is not involved in the proof). In order to handle the cases in which no such finite set of disequations can be generated, we define a loop detection rule, that encodes a form of mathematical induction (by "descente infinie") and that is able to derive clauses of the form $\alpha \not\approx t$ which cannot be derived by the superposition calculus. Intuitively, this rule applies when a cycle is detected in the search space, i.e. when S entails a set of $(\alpha, \mathfrak{C})$-clauses S' which is identical to S up to a shift of the value of the parameter α. The following definition formalizes this notion:

Definition 3. *A* shift *for a variable x is a substitution of the form $\{x \mapsto s\}$, where $s \neq x$ and $\mathit{var}(s) = \{x\}$. Let θ be a shift, C be a normalized $(\alpha, \mathfrak{C})$-clause and let t be a term with $\mathit{var}(t) = \{x\}$. The $(\alpha, \mathfrak{C})$-clause $\mathit{shift}(C, t, \theta)$ is defined as follows:*

– *If C is of the form $\alpha \not\simeq t\sigma \vee D$, for some substitution σ and for some clause $D \in \mathfrak{C}$, then* $\mathit{shift}(C, t, \theta) \stackrel{def}{=} \alpha \not\simeq t\theta\sigma \vee D$.
– *Otherwise* $\mathit{shift}(C, t, \theta) \stackrel{def}{=} C$.

If S is a set of $(\alpha, \mathfrak{C})$-clauses then $\mathit{shift}(S, t, \theta) \stackrel{def}{=} \bigcup_{C \in S} \{\mathit{shift}(C, t, \theta)\}$.

Example 2. Let $C : \alpha \not\simeq f(g(x)) \vee h_{g(x)}(y) \simeq y$ and $D : \alpha \not\simeq f(x) \vee h_x(y) \simeq a$. We have $\mathit{shift}(C, f(x), \{x \mapsto f'(x)\}) = \alpha \not\simeq f(f'(g(x))) \vee h_{g(x)}(y) \simeq y$, $\mathit{shift}(C, f(g(x)), \{x \mapsto f'(x)\}) = \alpha \not\simeq f(g(f'(x))) \vee h_{g(x)}(y) \simeq y$, $\mathit{shift}(D, f(x), \{x \mapsto f'(x)\}) = \alpha \not\simeq f(f'(x)) \vee h_x(y) \simeq a$ and $\mathit{shift}(D, f(g(x)), \{x \mapsto f'(x)\}) = D$.

The loop detection rule is based on the following theorem. Intuitively, it applies when, for all possible instances s of some term t, the branch in the search space that corresponds to the case $\alpha = s$ is either closed (i.e. the clause set contains a clause of the form $\alpha \not\simeq s'$, where $s' \succeq s$) or can be reduced (by shifting) to the branch corresponding to a strictly smaller term. Then the whole branch $\alpha = t$ can be closed, by "descente infinie".

A set of $(\alpha, \mathfrak{C})$-clauses S is a *t-set* if for every $(\alpha, \mathfrak{C})$-clause $\alpha \not\simeq s \vee C$ occurring in S, we have $s \preceq t$.

Theorem 2. *Let S be a set of $(\alpha, \mathfrak{C})$-clauses. Assume there exists a set of terms $\{t_1, \ldots, t_n\}$ covering for t such that:*

1. *For all $i \in [1, n]$, there exists a t_i-set $S_i \subseteq S$.*
2. *For all $i \in [1, n]$ and for all ground terms $s \preceq t_i$, one of the following conditions holds:*
   (a) $S_i \models \alpha \not\simeq s$.
   (b) There exist a number $j \in [1, n]$ and a shift θ_s such that $S_i \models \mathit{shift}(S_j, t_j, \theta_s)$ and $s \preceq t_j\theta_s$.

Then we have $S \models \{\alpha \not\simeq t\}$.

Theorem 2 allows one to derive an $(\alpha, \mathfrak{C})$-clause of the form $\alpha \not\simeq t$ from a set satisfying the previous properties. In practice, guessing the sets S_i and the shifts θ_s and checking whether (i) $S_i \models \alpha \not\simeq s$ or (ii) $S_i \models \mathit{shift}(S_j, t_j, \theta_s)$ is of course infeasible. We need to impose stronger syntactic conditions. A simple solution (used in the sequel) is to check that: (i) a clause of the form $\alpha \not\simeq s'$ with $s' \succeq s$ has been derived from parent clauses in S_i, (ii) $\mathit{shift}(S_j, t_i, \theta_s)$ has been derived from S_i. Of course, to apply the theorem in practice, one has to exhibit a *finite* set of substitutions θ_s that covers all possible terms, so that Condition 2 holds. The completeness proof of the next section provides additional hints on how the theorem should be applied in practice (in particular, to choose the sets S_i). From now on, we write $S \vdash C$ if an $(\alpha, \mathfrak{C})$-clause C can be deduced from a set of clause sets S by superposition or by the loop detection rule.

Example 3. Consider the set of $(\alpha, \mathfrak{C})$-clauses of Example 1. Let $t_1 = x$, $S_1 = \{1, 2, 3\}$ and $S' = \{1, 2, 5\}$. We check that the conditions of Theorem 2 are fulfilled. The set $\{t_1\}$ is obviously covering, and it is clear from the derivation in Example 1 that we have $S_1 \vdash^* \alpha \not\simeq 0$ and $S_1 \vdash^* S'$. Furthermore, it is easy to check that $S' = \mathit{shift}(S_1, x, \{x \mapsto$

$s(x)\}$). Let s be a ground term. The term s is either 0, in which case Condition 2.a is satisfied, or of the form $s(t')$ for some term t', in which case Condition 2.b holds (with $j = 1$ and $\theta_s = \{x \mapsto s(x)\}$). Thus Theorem 2 applies and the clause $\alpha \not\approx x$ is generated. This clause is obviously unsatisfiable, which proves that the original clause set is also unsatisfiable.

If the indices are natural numbers, the conditions are much simpler to test, since all covering sets contain a subset of the form $\{0, s(0), \ldots, s^k(0), s^{k+1}(x)\}$. Thus we may assume that $k = 1$ and that there exists at most one shift θ_s.

The loop detection rule is related to the inductive rule presented in [16]. Both rules apply globally, on the whole clause set (and not on a fixed finite set of premises, as the usual inference rules). They both encode a form of mathematical induction in the context of a superposition-based calculus, with the aim of deriving a contradiction in some cases where the other rules diverge. However, there exist important differences between these two rules. The inductive rule of [16] encodes the fact that, when proving a formula $\phi(\alpha)$, where α ranges over some inductively defined domain, one may assume that α is minimal, i.e. that $\neg\phi(\beta)$ holds for every β strictly lower than α (according to some well-founded ordering). The purpose of the inductive rule is precisely to derive the formula $\neg\phi(\beta)$, with additional constraints ensuring that $\beta < \alpha$. Obviously, this rule only preserves satisfiability. In contrast, our rule uses induction in the form of descente infinie: it only applies when a formula $\phi(\beta)$ has been *explicitly* derived from $\phi(\alpha)$ by the inference rules. The only properties that can be derived are clauses of the form $\alpha \not\approx t$, which in some sense close the branches corresponding to instances of t in the search space. The conclusion is a logical consequence of the premises, and the inference strongly depends on the considered signature.

Example 4. For instance, consider the clause set $\{\alpha \not\approx x \vee p_x, \neg p_a, \neg p_{f(x)} \vee q_x, \neg q_{f(x)} \vee q_x, \neg q_a\}$. The superposition calculus derives the clauses: $\alpha \not\approx a, \alpha \not\approx f^n(f(x)) \vee q_x$ and $\alpha \not\approx f^n(f(a))$, for every $n \in \mathbb{N}$. The inductive rule of [16] applies on the initial clause set and derives (for instance) the clause: $\alpha \not\approx f(x) \vee \neg p_x$. This is intuitively justified by the fact that if we assume that α is the minimal term such that $\alpha \not\approx x \vee p_x$ (i.e. p_α), holds then necessarily p_x cannot hold if x is a proper subterm of α. Notice that this clause does not help to derive a contradiction in this case. Our rule cannot derive such a property. However, since the clauses $\alpha \not\approx f(f(x)) \vee \neg p_x$ and $\alpha \not\approx f(a)$ can be derived from $\alpha \not\approx f(x) \vee p_x$ (using the clauses not containing α), and since $\alpha \not\approx f(f(x)) \vee \neg p_x = \mathit{shift}(\alpha \not\approx f(x) \vee p_x, f(x), \{x \mapsto f(x)\})$, the loop detection rule applies and derives $\alpha \not\approx f(x)$. Notice that this is only possible because the signature only contains f and a (otherwise the set $\{f(x), a\}$ would not be covering). Together with the clause $\alpha \not\approx a$, this proves the unsatisfiability of the initial clause set.

The loop detection rule also departs from the technique presented in [15] for deciding the validity of $\forall\exists$-queries. The latter approach is based, roughly speaking, on a "compilation" of the search space into an automaton, and to a reduction of the satisfiability problem to the emptiness problem for the represented language.

We show a more complex example of application of our approach.

Example 5. Let T be an array. Let $a, b_1, \ldots, b_n$ be indices of T, such that $\forall i\ a \neq b_i$. We consider the array T' obtained from T by changing successively the value of the cell b_i to some constant c_i. We want to prove that $T[a] = T'[a]$. This is formalized in our setting by the following set of clauses. We define a sequence of arrays T_i with the following clauses:

$$(1)\quad T_0 \simeq T \qquad (2)\quad T_{i+1} \simeq \mathit{store}(T_i, b_i, c_i)$$

We have the axiom:

$$(3)\quad b_i \not\simeq a$$

We also consider the usual axioms of the theory of arrays (see for instance [4]):

$$(4)\quad \mathit{select}(\mathit{store}(t, x, v), x) \simeq v \qquad (5)\quad x \simeq y \vee \mathit{select}(\mathit{store}(t, x, v), y) \simeq \mathit{select}(t, y)$$

The inequation $T_n[a] \neq T[a]$ is defined as follows:

$$(6)\quad \mathit{select}(T, a) \simeq d \qquad (7)\quad \alpha \not\approx i \vee \mathit{select}(T_i, a) \not\simeq d$$

We then derive the following $(\alpha, \mathfrak{C})$-clauses by applying the superposition calculus:

(8)	$\alpha \not\approx 0 \vee \mathit{select}(T, a) \not\simeq d$	(1,7)
(9)	$\alpha \not\approx 0$	(6,8)
(10)	$y \approx b_i \vee \mathit{select}(T_{i+1}, y) \simeq \mathit{select}(T_i, y)$	(2,5)
(11)	$\alpha \not\approx i+1 \vee a \simeq b_i \vee \mathit{select}(T_i, a) \not\simeq d$	(10,7)
(12)	$\alpha \not\approx i+1 \vee a \simeq b_i \vee \mathit{select}(T, a) \not\simeq d$	(1,11)
(13)	$\alpha \not\approx 1 \vee a \simeq b_0$	(6,12)
(14)	$\alpha \not\approx 1$	(13,3)
(15)	$\alpha \not\approx i+2 \vee a \simeq b_i \vee \mathit{select}(T_i, a) \not\simeq d$	(10,11)

We check that the conditions of Theorem 2 hold. Consider the sets $S_1 = \{1, 6, 10, 11\}$ and $S' = \{1, 6, 10, 15\}$. S_1 is an $i+1$-set and S' is an $i+2$-set. We have $S_1 \vdash^* S'$. Furthermore, $S_1 = \mathit{shift}(S', i+1, \{i \mapsto i+1\})$. The only ground term that is an instance of $i+1$ but not of $i+2$ is 1, and we have $S_1 \vdash^* \alpha \not\approx 1$. Hence the looping rule applies, yielding (16) $\alpha \not\approx i+1$. Since $\{0, i+1\}$ is covering, Clauses 9 and 16 entail that the original clause set is unsatisfiable. Notice that the loop detection rule can be applied in several different ways. In this case, if we consider the sets $S_1 = \{1, 6, 10, 11\}$, $S_2 = \{9\}$ and $S' = \{1, 6, 10, 15\}$, then one can *directly* generate $\alpha \not\approx i$ (since S_2 is a 0-set, $S_2 \vdash^* \alpha \not\simeq 0$ and $\{0, 1, i+2\}$ is covering).

5 Completeness

Since the considered logic is not semi-decidable (by Theorem 1), the proof procedure cannot be complete in general. That is why we impose some additional conditions on the considered clause sets. At this point, we only introduce abstract, semantic conditions which are meant to be as general as possible and sufficient to ensure completeness. In Section 7 we will provide an example of a concrete (syntactic) class of clause sets fulfilling these requirements.

For technical convenience, we assume that the considered terms contain no constant symbols of a sort in $\mathcal{S}_I$. This condition greatly simplifies the definitions

and proofs. It is not really restrictive, since any constant symbol a may be replaced by a term $a(x)$, where x is a dummy variable[1]. Furthermore, we assume that for every $(\alpha, \mathfrak{C})$-clause $\alpha \not\approx t \vee C$ occurring in the considered clause sets, t is not a variable. Again this condition is not restrictive: it can be enforced by introducing a new function symbol f of profile $s \to s'$, where s is the initial sort of α and s' is a new sort symbol, and by replacing every disequation $\alpha \not\approx t$ (where t is possibly a variable) by $\alpha \not\approx f(t)$ (note that by definition f is the unique function symbol of sort s', thus for all interpretations $\mathcal{I}$, $\mathcal{I}(\alpha)$ will be of the form $f(\ldots)$). A clause C is *index-flat* if every index occurring in C is a variable.

Definition 4. *We denote by succ the partial function such that* $succ(t) \stackrel{def}{=} f_1(\ldots(f_{n-1}(x)))$ *if* t *is of the form* $f_1(\ldots(f_n(x))\ldots))$ *for some variable* x *(succ is undefined otherwise).*

We now introduce a function "rank" that plays a central role in the following. It maps all $(\alpha, \mathfrak{C})$-clauses C to the (necessarily unique) term t such that $\alpha \not\approx t$ occurs in C (or $\bot$ if α does not occur in C). However, if C is not index-flat, then we will return not the term t itself, but rather its successor according to *succ*. The reason behind this seemingly non-intuitive definition is that we want the rank to be preserved by instantiation (for the clauses whose indices have depth 0 or 1), for instance $\alpha \not\approx f(g(x)) \vee p_{g(x)}$ should have the same rank as $\alpha \not\approx f(x) \vee p_x$. More formally:

Definition 5. *The* rank *of an* $(\alpha, \mathfrak{C})$*-clause* C *is defined as follows.*

- *If* $C \in \mathfrak{C}$ *then* $rank(C) \stackrel{def}{=} \bot$.
- *If* C *is of the form* $\alpha \not\approx i \vee D$ *and* D *is index-flat then* $rank(C) \stackrel{def}{=} i$.
- *If* C *is of the form* $\alpha \not\approx i \vee D$ *and* D *is not index-flat then* $rank(C) \stackrel{def}{=} succ(i)$.

The function rank is well-defined, since i cannot be a variable. If S is a set of $(\alpha, \mathfrak{C})$-clauses, we denote by $S\langle r\rangle$ the set of $(\alpha, \mathfrak{C})$-clauses of rank r (up to a renaming) in S. We then consider a particular subset of $\mathfrak{C}$, obtained by considering only the index-flat $(\alpha, \mathfrak{C})$-clauses that can interact with a non-index-flat $(\alpha, \mathfrak{C})$-clause:

Definition 6. *We denote by* $\mathfrak{F}$ *the set of index-flat clauses* $C \in \mathfrak{C}$ *such that there exist two clauses* $D, E \in \mathfrak{C}$ *such that* $C, D \vdash E$ *and* D *is not index-flat.*

If S is a set of $(\alpha, \mathfrak{C})$-clauses, we denote by $\mathfrak{F}(S)$ the set $\{\alpha \not\approx t \vee C \in S \mid C \in \mathfrak{F}\}$.

Definition 7. *The class* $\mathfrak{C}$ *is* admissible *iff it satisfies the following conditions:*

(c_1) $\mathfrak{C}$ *is closed under superposition i.e. if* $S \vdash C$ *and* $S \subseteq \mathfrak{C}$ *then* $C \in \mathfrak{C}$.

[1] Of course the signature Σ must contain a constant symbol of the same sort as x, otherwise the set of ground terms would be empty. However, this constant is not allowed to occur in the clauses.

(c_2) *Each non-empty clause in* $\mathfrak{C}$ *contains exactly one index variable. Furthermore, if* C *and* D *contain two index variables* x *and* y *respectively, and if an inference is applicable on* C, D *with a unifier* σ*, then* $\sigma(x)$ *is* x*,* y *or of the form* $f(y)$*, for some function symbol* f*. Moreover if* $\sigma(x) = f(y)$ *then* C *is index-flat and* D *is not. Similarly, if* C *contains an index variable* x*, and if a unary inference is applicable on* C *with a unifier* σ*, then we must have* $\sigma(x) = x$*.*

(c_3) $\mathfrak{F}$ *is finite (up to a renaming of variables).*

From now on we assume that the considered class $\mathfrak{C}$ is admissible. Condition (c_1) is rather natural: it guarantees that the $(\alpha, \mathfrak{C})$-clauses constructed over $\mathfrak{C}$ are closed under superposition inferences. Condition (c_2) will ensure that these inferences cannot increase the depth of the indices arbitrarily. Condition (c_3) ensures that only finitely many clauses of a given rank can be built on $\mathfrak{F}$. As we will show, (c_2) entails that the search space has a strict hierarchic structure w.r.t. the rank: for all terms $t \succeq s$, the $(\alpha, \mathfrak{C})$-clauses of rank s are necessarily derived from those of rank t (along with the clauses of rank $\bot$). Furthermore, we will prove that this relation still holds if the clauses are restricted to those occurring in $\mathfrak{F}$, i.e. we have $t \succeq s \Rightarrow S\langle\bot\rangle \cup \mathfrak{F}(S\langle t\rangle) \vdash^* S\langle\bot\rangle \cup \mathfrak{F}(S\langle s\rangle)$ (assuming t is lower than the ranks of the initial $(\alpha, \mathfrak{C})$-clauses). Then, (c_3) ensures that there exist only finitely many sets $\mathfrak{F}(S\langle t\rangle)$ (up to a shift), which entails that the loop detection rule eventually applies. A clause set S is *saturated* if every clause C such that $S \vdash C$ is redundant w.r.t. S (in the usual sense). The following theorem states our completeness result:

Theorem 3. *Let* S *be a set of* $(\alpha, \mathfrak{C})$*-clauses where* $\mathfrak{C}$ *is admissible. If* S *is saturated and unsatisfiable then either* $\square$ *or* $\alpha \not\approx x$ *occurs in* S*.*

6 Satisfiability Detection

In standard clausal logic, the satisfiability of a given clause set S can sometimes be established by saturation, in case the set of clauses derived from S is finite and does not contain $\square$. This is not feasible in the context of this paper (except in trivial cases), because the rank will in general increase indefinitely. However, we can devise the following satisfiability test, based on a form of *partial* saturation:

Definition 8. *A set of* $(\alpha, \mathfrak{C})$*-clauses* S *is* saturated w.r.t. a (ground) term t *if for every clause* C *such that* $S \vdash C$*, either* C *is redundant or* C *is of the form* $\alpha \not\approx s \vee D$*, where* s *and* t *are not unifiable.*

If a set of $(\alpha, \mathfrak{C})$-clauses S is saturated w.r.t. some term t and does not contain $\alpha \not\approx t$, then it can be shown that S has a model in which the value of α is t (note that testing whether a clause set is saturated w.r.t. t is easy: if the standard proof search algorithm is used, it suffices to test that the set of "active" clauses contains no clause of a rank more general than t).

If $\mathfrak{C}$ is finite (i.e. if the superposition calculus terminates on clause sets in $\mathfrak{C}$), then the number of $(\alpha, \mathfrak{C})$-clauses of a fixed rank is also finite. This entails that, for every ground term t, it is possible to eventually obtain, from any initial

clause set S, a clause set containing S and saturated w.r.t. t. If, moreover, there exists a term t such that this partially saturated set does not contain $\alpha \not\simeq t$, then satisfiability can be detected. If no such term exists, the initial clause set must be unsatisfiable, thus termination can be ensured in both cases, although the set of $(\alpha, \mathfrak{C})$-clauses is infinite.

Theorem 4. *If $\mathfrak{C}$ is finite, then the satisfiability problem is decidable for sets of $(\alpha, \mathfrak{C})$-clauses.*

Note that the fact that $\mathfrak{C}$ is finite does not imply that the set of terms that can be constructed on the signature is finite, since the restrictions on the superposition inferences can prevent such terms from being generated.

7 Complete Classes of Indexed Formulæ

In this section, we demonstrate the applicability of our results by providing an example of an admissible *syntactic* class of $(\alpha, \mathfrak{C})$-clauses.
Let μ be a *complexity function* mapping every ground term to a natural number. We assume that for any $k \in \mathbb{N}$, the set of ground objects t such that $\mu(t) \leq k$ and such that every index in t is of depth at most 1 is finite. Examples of usual complexity functions satisfying this requirement include the depth (maximal length of the non-index positions occurring in the terms) or the size (number of non-index positions in the terms). The function μ is extended to atom, literals and clauses as follows:

- $\mu(t \not\simeq s) \stackrel{\text{def}}{=} \mu(t \simeq s) \stackrel{\text{def}}{=} \max(\mu(t), \mu(s))$, and
- $\mu(\bigvee_{i=1}^{n} l_i) \stackrel{\text{def}}{=} \max\{\mu(l_i) \mid i \in [1, n]\}$.

We write $t =_\mu s$ if for every substitution σ we have $\mu(t\sigma) = \mu(s\sigma)$. For every natural number ν, we write $t \leq_\mu^\nu s$ if for every substitution σ such that $\mu(t\sigma) > \nu$, we have $\mu(t\sigma) \leq \mu(s\sigma)$. The relations $=_\mu$ and $\leq_\mu^\nu$ are hard to test in general because the set of substitutions σ is infinite. However, algorithms for testing whether $t =_\mu s$ or $t \geq_\mu^\nu s$ for various complexity functions μ are defined in [19, 21]. For instance, if μ is the depth of the term, then it is easy to see that $t =_\mu s$ iff the following conditions hold: $\mu(t) = \mu(s)$, t and s have the same set of variables and the maximal occurrence depth of every variable is the same in t and in s.

We consider two (not necessarily disjoint) sets of predicate symbols: a set of *control predicates* Ω_c and a set of *index propagation predicates* Ω_i. A literal is a *control literal* (resp. an *index propagation literal*) if its atom is of the form $p_i(t_1, \ldots, t_n) \simeq \mathtt{true}$, where $p \in \Omega_c$ (resp. $p \in \Omega_i$) and i is an index term. The remaining literals are called the *standard literals*. Let $\mathcal{S}_p$ be a set of sort symbols, called the *μ-preserving sorts* satisfying the following properties:

- For every predicate symbol $p \in \Omega_c \cup \Omega_i$ of profile $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{bool}$, and for every $i \in [1, n]$, we have $\mathtt{s}_i \in \mathcal{S}_p$.
- For every function symbol of profile $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{s}$, if $\mathtt{s} \in \mathcal{S}_p$ then $\forall i \in [1, n], \mathtt{s}_i \in \mathcal{S}_p$.

Intuitively, the sorts in $\mathcal{S}_p$ are the sorts of the terms occurring in control or index propagation literals.

Definition 9. *A set of clauses S is* μ-controlled *iff the following conditions holds:*

1. *For every equation $t \simeq s$ occurring in a clause in S, if t and s are of a sort in $\mathcal{S}_p$, then $t =_\mu s$.*
2. *There exists a natural number ν such that, for every clause C, and for every index propagation literal or positive control literal L in C, we have $L \leq^\nu_\mu C'$, where C' is the set of negative control literals occurring in C.*
3. *All literals, except index propagation literals, are index-flat.*
4. *Every clause in S contains at most one index variable.*
5. *The atoms in S are either of the form $f_i(\boldsymbol{t}) \simeq g_i(\boldsymbol{s})$ (equational atoms) or of the form $p_i(\boldsymbol{t}) \simeq \mathtt{true}$ (non-equational atoms).*

In particular, any non-equational index-flat clause set is μ-controlled (with $\mathcal{S}_p = \Omega_c = \Omega_i = \emptyset$). Condition 1 ensures that the superposition inferences will not affect the complexity of the terms occurring inside control or index propagation literals. Condition 2 states that the complexity of every positive control literal and of every index propagation literals is dominated by the complexity of the negative control literals occurring in the clause. Condition 3 ensures that the only dependencies between terms of distinct indices are encoded by index-propagation literals (the remaining literals state relations involving only terms with the same indices). Condition 5 forbids equations between indexed and non-index non-boolean terms, such as $a_i \simeq b$. Equations between variables are also forbidden.

Example 6. Let μ be the depth function. Let $\Omega_c = \{p\}$, $\Omega_i = \{q\}$. The clauses $p_i(f(c))$, $\neg p_i(x) \vee p_{s(i)}(x) \vee \neg q_i(x)$, $p_a(b)$, $\alpha \not\simeq i \vee p_i(f(b))$, $q_i(x) \vee \neg q_i(f(x))$, $q_i(f(c))$, $p_i(x) \vee \neg r_i(y) \vee r_i(f(y))$, $r_i(a)$, are μ-controlled, with $\nu = 2$. Note that the conditions on the control literals ensure that for all literals of the form $p_i(t)$ or $q_i(t)$ generated by the inferences, t is of depth at most 2. In contrast, the literals of head r_i can be of arbitrary depth. The clauses $f(x) \simeq g(f(x))$, $\neg q_i(x) \vee q_i(f(x))$, $p_i(f(x)) \vee \neg q_i(x)$, $p_i(x)$, $a_{s(i)} \simeq b_i$, $p_i \vee p_j$, $f(x) \simeq g_i(x)$ are not μ-controlled, because they contradict Conditions $1, 2, 2, 2, 3, 4$ and 5, respectively. In particular, $f(x) \simeq g(f(x))$ contradicts Condition 1 because $f(x)$ and $g(f(x))$ have distinct depths. Similarly, $\neg q_i(x) \vee q_i(f(x))$ violates Condition 2 (regardless of the value of ν), because the depth of $q_i(x)$ is not asymptotically greater than that of $q_i(f(x))$.

The superposition strategy is defined as follows.

- Negative control literals are selected in any clause containing only control literal and index propagation literals. Otherwise, the selected literals are the maximal ones.
- The ordering satisfies the following properties:
  - The relation $p_{f(i)}(t_1, \ldots, t_n) > q_i(s_1, \ldots, s_m)$ holds for all symbols $p, q, t_1, \ldots, t_n, s_1, \ldots, s_m$ and f.
  - All standard atoms of index i are strictly greater than all index propagation atoms with the same index i.

The second condition on the ordering may seem rather strong, since, clearly, the considered index propagation atom can contain variables not occurring in the standard atom. However, it can easily be enforced by assuming that the terms occurring at a root level in the standard atoms are of some special sorts that cannot occur inside an index propagation atom (those terms can then be assumed to be strictly greater than the ones occurring in index propagation atoms).

Theorem 5. *The class of μ-controlled clauses is admissible.*

Proof. We prove simultaneously that Conditions c_1, c_2 and c_3 holds. In particular, c_3 is established by showing that every clause in $\mathfrak{F}$ contains no variable except index variables and is of complexity lower than ν (this clearly entails that $\mathfrak{F}$ is finite up to a renaming). We consider two μ-controlled clauses $C[t]_p$ and $u \simeq v \vee D$ and a clause $C[v]_p\sigma \vee D\sigma$, obtained by superposition from the two first clauses (the proof for the other inference rules is similar).

- Assume that $C[t]_p$ is index-flat and that $u \simeq v \vee D$ is not. Due to the ordering used to restrict the inferences, u cannot be index-flat (otherwise u would not be maximal). Therefore, u is of the form $p_{f(i)}(\boldsymbol{t})$, where $p \in \Omega_i$. Consequently, t is of the form $p_j(\boldsymbol{s})$ (since t and u are unifiable, they must have the same head symbol). But then since t is selected, $C[t]_p$ cannot contain any standard atom (otherwise t would not be maximal) neither any negative control literal (otherwise this literal would be selected). Thus the set of negative control literals in C is empty, and by Condition 2 in Definition 9, we have $\mu(C[t]_p) \leq^{\nu}_{\mu} \Box$. This implies that $C[t]_p$ contains no variables (except index variables) and is of complexity at most ν. The same reasoning holds if $u \simeq v \vee D$ is index-flat and $C[t]_p$ is not (in this case $u \simeq v \vee D$ contains no non-index variable and we must have $\mu(u \simeq v \vee D) \leq \nu$). Thus the clauses in $\mathfrak{F}$ contain no variable (except index variables) and are of complexity at most ν.
- Since the depth of the index terms is at most 1 and since, due to the ordering restrictions, only the literals with deepest indices are eligible for inferences, the condition c_2 is easy to check. Notice that this implies that the number of index variables does not increase.
- It only remains to prove that c_1 holds, i.e. that $C[v]_p\sigma \vee D\sigma$ is μ-controlled. We prove that this clause satisfies all the conditions of Definition 9.
  - Let $t \simeq s$ be an equation in $C[v]_p\sigma \vee D\sigma$, where t, s are of a sort in $\mathcal{S}_p$. If $t \simeq s$ is of the form $(t' \simeq s')\sigma$ where $t' \simeq s'$ occurs in one of the parent clauses, then since the parent clauses are μ-controlled by hypothesis, we have necessarily $t' =_\mu s'$ and thus also $t =_\mu s$. Otherwise, $t \simeq s$ is of the form $(t'[v]_q \simeq s')\sigma$, where $t' \simeq s'$ occurs in one of the parent clauses and $t'|_q = u$. Again, we have $t' =_\mu s'$. Furthermore, by definition of $\mathcal{S}_p$, u and v must be of a sort in $\mathcal{S}_p$. Consequently, we have also $u =_\mu v$ and thus $t' =_\mu t'[v]_q$. Therefore, $t'[v]_q\sigma =_\mu s'\sigma$.
  - Let L be a literal occurring in $C[v]_p\sigma \vee D\sigma$, that is either an index propagation literal or a positive control literal. By definition, L is obtained

from a literal L' occurring in one of the parent clauses by applying the substitution σ and by (possibly) replacing an occurrence of the term $u\sigma$ by $v\sigma$. If u does not occur in L then obviously $L = L'\sigma$, thus $L =_\mu L'\sigma$. If u occurs in L then by definition of $\mathcal{S}_p$, u must be of a sort in $\mathcal{S}_p$, thus we have $u =_\mu v$ and therefore in both cases the relation $L =_\mu L'\sigma$ holds. Since the parent clauses containing L' is μ-controlled, it also contains a disjunction of control literals M such that $M \geq^\nu_\mu L'$. Thus $C[v]_p\sigma \vee D\sigma$ contains a disjunction of literals M' obtained from $M\sigma$ by replacing $u\sigma$ by $v\sigma$. If $u \simeq v$ is a control literal, then D contains a disjunction of negative control literals D' such that $D' \geq^\nu_\mu u \simeq v$. Thus we have $D'\sigma \geq^\nu_\mu M\sigma \geq^\nu_\mu L$. Otherwise, if u occurs in M then it must be of a sort in $\mathcal{S}_p$. Therefore we have $u =_\mu v$, which implies that $M' =_\mu M\sigma$, and thus $M' \geq^\nu_\mu L$.

- Assume that $C[v]_p\sigma \vee D\sigma$ contains a literal L that is not an index propagation literal and that is not index-flat. This literal is obtained from a literal L' occurring in one of the parent clauses by applying the substitution σ and by replacing the term $u\sigma$ by $v\sigma$. L' cannot be an index propagation literal . Thus L' is index-flat, which means that σ cannot be flat and that (by Condition c_2) the parent clause not containing the literal L' must be non-index-flat. But we have shown that the only index-flat clauses that can interact with non-index-flat clauses only contain index propagation literals, which contradicts our initial assumption on L.
- Condition 4 is an immediate consequence of c_2.
- The last condition is straightforward to check, since it holds for the two parents and it is obviously preserved by replacement and instantiation.

Intuitively, the index propagation literals encode properties of the index terms and relations between them, whereas the other literals encode properties of standard terms (for a given index). The use of control literals ensures that the size of the former literals is bounded whereas the latter can be arbitrary first-order literals. Several concrete classes can be obtained simply by instantiating the complexity measure μ. All these classes are strictly more expressive than first-order logic (which corresponds to the case in which both Ω_c and Ω_i are empty). Theorem 3 ensures that our calculus is complete for μ-controlled clause sets. If, moreover, the considered clauses belong to a class for which the superposition calculus terminates (such as the monadic class, the guarded fragment,... or if all the literals are index propagation or control literals) then Theorem 4 ensures termination and decidability. We thus obtain – without any additional effort – a general completeness result for schemata of first-order clauses and decidability results for schemata built on decidable subclasses of first-order logic. In particular, it is easy to check (see [3]) that every regular propositional schema [1] can be expressed as a set of μ-controlled clauses (in this case all literals are index propagation literals and indices are natural numbers). Therefore, the formulæ of propositional linear temporal logic can also be encoded as μ-controlled clause sets (see [2] for a translation of LTL into regular schemata). Many properties of usual inductively defined data-structures such as lists or trees can be encoded

into controlled clauses. For instance, a tree can be denoted as a constant symbol τ indexed by positions[2], where τ_p denotes the label of the node at position p. Then we can easily encode the fact that some first-order property is satisfied by all labels in the tree (or by all labels occurring along some position, or some regular set of positions). However, we *cannot* express the fact that, e.g., a tree is sorted, because it requires to use atoms of the form $\tau_{p.0} \leq \tau_p \leq \tau_{p.1}$, which necessarily involve terms with several *distinct* indices. Relations between distinct trees can also be expressed, provided they preserve the shape of the tree (for instance we can state that a tree is obtained from τ by applying some function f on every node).

8 Conclusion

A proof procedure for handling clauses with indexed terms has been presented, enriching the superposition calculus with a carefully controlled form of inductive reasoning. Although the satisfiability problem is not even semi-decidable in general, criteria have been devised to ensure refutational completeness and termination. At the best of our knowledge, no other proof procedure provides similar features. Future work includes the implementation of this procedure and the evaluation of its practical performance. To this purpose, developing efficient algorithms to apply the loop detection rule is obviously essential (Sections 4 and 5 provide some hints in this direction). A first implementation has already been completed in the particular case in which the indices are natural numbers (defined on the signature $\{0, succ\}$) and will soon be available. From a more theoretical point of view, other classes of clause sets satisfying the conditions of Section 5 have to be identified. In particular, it would be interesting to find a terminating class containing the example provided in Section 4 concerning the theory of arrays (as well as examples from other similar theories: lists, records, integers etc.). Another line of future work is to extend the proof procedure in order to allow equations between indices.

References

1. V. Aravantinos, R. Caferra, and N. Peltier. Decidability and undecidability results for propositional schemata. *Journal of Artificial Intelligence Research*, 40:599–656, 2011.
2. V. Aravantinos, R. Caferra, and N. Peltier. Linear Temporal Logic and Propositional Schemata, Back and Forth. In *TIME'2011 (18th International Symposium on Temporal Representation and Reasoning)*, 2011.
3. V. Aravantinos, M. Echenim, and N. Peltier. A resolution calculus for first-order schemata. *Fundamenta Informaticae*, 2013. Accepted for publication, to appear.
4. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
5. M. Baaz, S. Hetzl, A. Leitsch, C. Richter, and H. Spohr. CERES: An analysis of Fürstenberg's proof of the infinity of primes. *Theor. Comput. Sci.*, 403(2-3):160–175, 2008.
6. L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[2] It is clear that positions can be encoded in a monadic signature.

7. L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierachic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.
8. D. Baelde, D. Miller, and Z. Snow. Focused inductive theorem proving. In *IJCAR*, pages 278–292, 2010.
9. 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.
10. A. Bundy. The automation of proof by mathematical induction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 845–911. Elsevier and MIT Press, 2001.
11. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, New York, NY, USA, 2005.
12. H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 14, pages 913–962. North-Holland, 2001.
13. H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–475, 1989.
14. A. Gupta and A. L. Fisher. Parametric circuit representation using inductive boolean functions. In C. Courcoubetis, editor, *CAV*, volume 697 of *LNCS*, pages 15–28. Springer, 1993.
15. M. Horbach and C. Weidenbach. Deciding the inductive validity of for all there exists * queries. In E. Grädel and R. Kahle, editors, *CSL*, volume 5771 of *LNCS*, pages 332–347. Springer, 2009.
16. M. Horbach and C. Weidenbach. Superposition for fixed domains. *ACM Trans. Comput. Logic*, 11(4):1–35, 2010.
17. D. Kapur and D. Musser. Proof by consistency. *Artificial Intelligence*, 31, 1987.
18. A. Kersani and N. Peltier. Completeness and Decidability Results for First-order Clauses with Indices (long version), Research Report, 2013. `http://membres-lig.imag.fr/peltier/kp13.pdf`.
19. A. Leitsch. Deciding clause classes by semantic clash resolution. *Fundamenta Informaticae*, 18:163–182, 1993.
20. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
21. N. Peltier. Some Techniques for Proving Termination of the Hyperresolution Calculus. *Journal of Automated Reasoning*, 35:391–427, 2006.
22. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.