Noname manuscript No.
(will be inserted by the editor)

An Instantiation Scheme for Satisfiability Modulo Theories

Mnacho Echenim · Nicolas Peltier

May 2010

Abstract State-of-the-art theory solvers generally rely on an instantiation of the axioms of the theory, and depending on the solvers, this instantiation is more or less explicit. This paper introduces a generic instantiation scheme for solving SMT problems, along with syntactic criteria to identify the classes of clauses for which it is complete. The instantiation scheme itself is simple to implement, and we have produced an implementation of the syntactic criteria that guarantee a given set of clauses can be safely instantiated. We used our implementation to test the completeness of our scheme for several theories of interest in the SMT community, some of which are listed in the last section of this paper.

1 Introduction

Most formal verification tools rely on procedures that decide the validity or, dually, the satisfiability of logical formulas. In general, the considered formula (or set of clauses) is ground and its validity needs only be tested *modulo* a background theory $\mathcal{T}$. In formal software verification for example, the background theory can define one or a combination of data structures such as arrays or lists. These problems are known as $\mathcal{T}$-*decision problems* or more commonly, *SMT problems*, and the tools capable of solving these problems are known as $\mathcal{T}$-*decision procedures*, or *SMT solvers* (SMT stands for *Satisfiability Modulo Theories*).

A lot of research has been devoted to the design of SMT solvers that are both efficient and scalable. Generally, state-of-the-art SMT solvers rely on algorithms based on the DPLL procedure [14,13] to deal with the boolean part of the SMT problems. These solvers deal with the theory reasoning by applying methods ranging from the *eager approach*, which consists in applying sophisticated techniques to reduce the entire SMT problem to an equisatisfiable SAT problem, to the *lazy approach*, which consists in searching for a conjunction of literals satisfying the boolean part of the formula,

Mnacho Echenim
Université de Grenoble
E-mail: Mnacho.Echenim@imag.fr

Nicolas Peltier
Université de Grenoble, CNRS
E-mail: Nicolas.Peltier@imag.fr

and testing whether this conjunction of literals is satisfiable modulo the background theory. A survey on SMT solvers and the different approaches can be found in [6].

Quantifier reasoning is meant to tackle the issue of solving SMT problems with formulas involving quantifiers. Most approaches rely on the original work of [17] on the Simplify prover, in which heuristics for quantifier instantiation are devised. State-of-the-art techniques include [22,16,30]. Of course, these heuristics are not complete in general, and the class for which completeness is ensured is not precisely characterized. Complete quantifier elimination can be efficiently performed for some particular theories, such as linear arithmetic (see for instance [18]) or a subclass of the theory of arrays [12].

The rewrite-based approach to solving SMT problems, initiated by [3], consists in employing a theorem prover for first-order logic with equality to solve the SMT problems. This approach is appealing, since by feeding any finitely axiomatized theory along with a set of ground clauses, one obtains a system that is refutationally complete. If the theorem prover is guaranteed to terminate on the input set, then it acts as a decision procedure for the background theory. Thus, much research on this subject is devoted to determining results on termination for the theorem prover [2,10,29,8]. The main issue with the rewrite-based approach is that theorem provers are not designed to handle the boolean part of a formula as efficiently as possible. Therefore, they do not perform well on SMT problems with a large boolean part. A solution to this problem consists in integrating the theorem prover with a DPLL-based tool, and although this raises new issues, such an integration was accomplished in, e.g., [32,33,15].

Another solution to this problem was investigated in [9], which consists in decomposing the SMT problem into a *definitional* part, made up of a conjunction of ground literals, and an *operational* part, containing the boolean structure of the problem. During the first stage, the theorem prover is fed the definitional part of the SMT problem along with the background theory, and the saturation process *compiles* the theory away. During the second stage, the saturated set generated by the theorem prover is fed to a DPLL-based tool, along with the operational part of the SMT problem. This approach allows to exploit the full power of the theorem prover and the DPLL-based tool, without requiring a tight integration between them. However, in the theory of arrays for example, the set of clauses obtained after compiling the theory away is not ground, and requires an additional instantiation before being fed to the DPLL-based tool.

The goal of this paper is to investigate how the instantiation phases of the saturation process can be singled out in order to devise a generic instantiation scheme for solving SMT problems. Solving an SMT problem modulo a background theory for which the instantiation scheme is complete would thus reduce to testing the satisfiability of a ground formula in first-order logic with equality, with no mention to any background theory. This scheme is meant to be as efficient as possible, i.e., instantiate the non-ground clauses under consideration as little as possible.

This approach is close to that of [12] for the theory of arrays. However, contrary to that of [12] which is model-theoretic, this one is proof-theoretic, and is not restricted to just one theory or its extensions. The efficiency requirement comes at the expense of completeness, and contrary to the schemes of [21,31,24], it is not always guaranteed that the original set of clauses and the instantiated one are equisatisfiable. However, the class of theories for which the scheme is complete is large enough to capture several theories of interest in the SMT community.

This paper presents the instantiation scheme that was devised, along with two sets of syntactic criteria on clauses, that guarantee the instantiated set of clauses and the original one are equisatisfiable. The first set of criteria is simpler to implement, and the other is more general. The imposed conditions have a common characteristic: they are based on syntactic properties of the arguments of function symbols. Depending on their positions, the former may be required to be ground, or to have a limited depth. These conditions come up quite naturally in our proof-theoretic setting, and are general enough for many theories of data structures from SMT problems to comply by them. These criteria have been implemented[1], and have allowed us to verify automatically that the instantiation scheme can be applied to several theories such as arrays, records or lists.

The rest of the paper is organized as follows.

- Section 2 reviews some usual definitions and basic results on equational clausal logic and superposition-based theorem proving. Most definitions are standard although we introduce some additional notations that are useful in our context. We also introduce two classes of terms that play a central rôle in the paper: the classes of I_0-flat and I_{nv}-closed terms, and several properties of these classes are proven.
- In Section 3 a new instantiation scheme is devised which reduces a set of first-order clauses S to a set of ground instances $\hat{S}_\lambda$. $\hat{S}_\lambda$ is always finite, which implies that, in contrast to other approaches (e.g. [28, 31, 21]), our instantiation scheme is not complete (S and $\hat{S}_\lambda$ are not always equisatisfiable). Some simple semantic conditions are proposed in order to ensure completeness. These conditions are proof-theoretic: the instantiation scheme is complete if S admits a particular kind of refutation, called a *simple* refutation. Clause sets admitting simple refutations are called *simply provable clause sets*.
- In Section 4 we introduce the syntactic class of so-called *controlled* clause sets, and we prove that they are always simply provable, which implies that the satisfiability problem is decidable for controlled sets.
- Controlled clause sets are not expressive enough to include usual theories of interest such as the theory of arrays. Section 5 shows that the conditions of Section 4 can be relaxed to handle a larger class of theories. The basic idea is that we only need to ensure that the relevant *consequences* of the considered clause set will be controlled, even if the parent clauses are not controlled. The clause sets satisfying these conditions are called *controllable clause sets*.
- Section 6 contains examples of controllable theories (including all the theories that can be handled by the superposition-based approach), additional algorithms and practical remarks.
- Finally, Section 7 concludes the paper.

Due to the length of the paper, some of the proofs (in particular the most technical ones) have been shifted to the appendix in order to improve readability. Also, our results extend in a straightforward manner to a many-sorted framework, but we have chosen to present them in a single-sorted framework for the sake of clarity.

[1] http://membres-lig.imag.fr/peltier/fish.html

2 Preliminaries

2.1 Basic definitions

In this section we briefly review some usual definitions and notations in Logic and Automated Theorem Proving. The results are standard and their proofs are omitted; we refer the reader to, e.g., [5,4] for details.

We assume given a set of *variables* $\mathcal{V}$, a set of *function symbols* Σ (containing at least a constant symbol) and an *arity function* mapping each element of Σ to a natural number. Σ_n denotes the set of symbols of arity n in Σ. Throughout this paper, a, b, c always denote constant symbols, f, g, h denote function symbols and x, y, z denote variables (possibly with indices). The symbol `true` denotes a special constant symbol used to encode predicate symbols.

The sets of *terms*, *atoms*, *literals* and *clauses* are defined in the standard way on Σ and $\mathcal{V}$, using the equality symbol $\simeq$. The notions of *interpretations*, *models* and *satisfiability* are defined as usual. Two sets of clauses S, S' are *equisatisfiable* if either S, S' are both unsatisfiable or S, S' are both satisfiable.

A *substitution* is a function mapping every variable to a term. The set of variables x such that $x\sigma \neq x$ is called the *domain* of σ and denoted by $dom(\sigma)$. A substitution σ of domain $x_1, \ldots, x_n$ such that $x_i\sigma = t_i$ for $i = 1, \ldots, n$ is usually denoted by $\{x_i \mapsto t_i \mid i \in [1..n]\}$. As usual, a substitution can be extended into a homomorphism on terms, atoms, literals and clauses. The image of an expression e by a substitution σ will be denoted by $e\sigma$. If E is a set of expressions, then $E\sigma$ denotes the set $\{e\sigma \mid e \in E\}$. The composition of two substitutions σ and θ is denoted by $\sigma\theta$. A substitution σ is *more general* than θ if there exists a substitution η such that $\theta = \sigma\eta$. The substitution σ is a *renaming* if it is injective and $\forall x \in dom(\sigma), x\sigma \in \mathcal{V}$; and it is a *unifier* of two terms t, s if $t\sigma = s\sigma$. Any unifiable pair of terms (t, s) has a most general unifier, unique up to a renaming, and denoted by $mgu(t, s)$. A term or clause containing no variable is *ground*. A substitution σ is *ground* if $x\sigma$ is ground, for every variable x in its domain.

A *position* is a finite sequence of natural numbers. ϵ denotes the empty sequence and $p.q$ denotes the concatenation of p and q. p is a *position in t* if either $p = \epsilon$ or $p = i.q$, $t = f(t_1, \ldots, t_n)$ and q is a position in t_i. $t|_p$ and $t[s]_p$ respectively denote the subterm at position p in t and the term obtained by replacing the term at position p by s: $t|_\epsilon \stackrel{\text{def}}{=} t$, $f(t_1, \ldots, t_n)|_{i.q} \stackrel{\text{def}}{=} t_i|_q$, $t[s]_\epsilon \stackrel{\text{def}}{=} s$ and $f(t_1, \ldots, t_n)[s]_{i.q} \stackrel{\text{def}}{=} f(t_1, \ldots, t_{i-1}, t_i[s]_q, t_{i+1}, \ldots, t_n)$. These notions extend straightforwardly to atoms, literals or clauses.

Flatness

A term is *flat* if it is a variable or a constant symbol. The set of flat terms is denoted by T_0: $T_0 \stackrel{\text{def}}{=} \mathcal{V} \cup \Sigma_0$. A non-flat term is *complex*. A clause C is *flat*[2] if for every literal $t \simeq s$ or $t \not\simeq s$ occurring in C, $t, s \in T_0$. A substitution σ is *flat* if $\forall x \in \mathcal{V}, x\sigma \in T_0$. A clause C is a *flat instance* of D if there exists a flat substitution σ such that $C = D\sigma$; note that C is not necessarily flat. Flat substitutions are stable by composition:

Proposition 1 *Let σ and μ be flat substitutions. Then $\sigma\mu$ is also flat.*

Any set of ground clauses can be *flattened*, by introducing fresh constants that serve as names for complex terms. This operation produces a set of ground clauses such that the only non-flat clauses are of the form $f(a_1, \ldots, a_n) \simeq b$, for constants $a_1, \ldots, a_n, b$.

[2] Note that we depart from the definition of, e.g., [3], where a flat literal can be of the form $f(a_1, \ldots, a_n) \simeq b$ for some flat terms $a_1, \ldots, a_n, b \in T_0$.

For example, $S = \{f(a) \not\simeq f(c) \vee a \simeq c,\ f(b) \not\simeq f(c) \vee b \simeq c\}$ is flattened by introducing the fresh constants a', b' and c', and replacing S by

$$\{f(a) \simeq a',\ f(b) \simeq b',\ f(c) \simeq c',\ a' \not\simeq c' \vee a \simeq c,\ b' \not\simeq c' \vee b \simeq c\}.$$

The original set of ground clauses and the flattened one are equisatisfiable.

2.2 Superposition

We review some basic notions about superposition-based theorem proving, following the formalism of [5].

Selection and Inference

Let $<$ denote a reduction ordering which is substitution-monotonic (i.e. for every σ, $(t < s) \Rightarrow (t\sigma < s\sigma)$). We assume that if a is a constant and t is a complex term, then $a < t$. This property on orderings is termed as the *goodness* property in [2]. The ordering $<$ is extended to atoms, literals and clauses using the multiset extension. A literal L is *maximal* in a clause 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. A term t is *eligible* in a clause C if t is not a variable and there exist two terms u, v such that $u \not< v$, t occurs in u and $\mathrm{sel}(C)$ contains either $u \simeq v$ or $u \not\simeq v$.

Definition 1 A clause C is *variable-eligible* if $\mathrm{sel}(C)$ contains a literal of the form $x \simeq t$ or $x \not\simeq t$, where $x \in \mathcal{V}$ and $x \not< t$.

For example, assume that $\mathrm{sel}(C)$ is the set of maximal literals in clause C. Then $(x \simeq y) \vee (f(z) \simeq a)$ and $(x \simeq a) \vee (y \simeq b)$ are variable-eligible whereas the clause $(f(x) \simeq x) \vee (y \not\simeq z) \vee (g(y,z) \simeq a)$ is not.

This notion is strongly related to the one of variable-inactive clauses that is defined in [2]. The properties of the reduction ordering we consider imply that:

Proposition 2 *Every flat clause is either ground or variable-eligible.*

Redundancy

A *tautology* is a clause containing two complementary literals, or a literal of the form $t \simeq t$. A clause C is *subsumed* by a clause D if there exists a substitution σ such that $D\sigma \subseteq C$. A ground clause C is *redundant* in S if there exists a set of clauses S' such that $S' \models C$, and for every $D \in S'$, D is an instance of a clause in S such that $D < C$. A non ground clause C is *redundant* if all its instances are redundant. In particular, every (strictly) subsumed clause and every tautological clause is redundant.

In practice, one has to use a decidable approximation of this notion of redundancy. We will assume that a clause C is redundant if it can be rewritten (using equational axioms) to either a tautology or to a subsumed clause (all the clauses satisfying this property are redundant in the previous sense).

The following stability result holds for redundant clauses:

Proposition 3 *If a clause C is redundant w.r.t. a set of clauses $S \cup \{D\}$ and if D is redundant w.r.t. S, then C is redundant w.r.t. S.*

Superposition calculus:

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

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

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

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

Fig. 1 The superposition calculus

Derivation Relations and Saturated Sets

We consider the calculus (parameterized by $<$ and sel) of Figure 1. If S is a set of clauses, we write $S \rightarrow^{\sigma}_{\text{sel},<} C$ when C can be deduced from S in one step by applying one of the rules of Figure 1, using the m.g.u. σ. We denote by $\mathcal{S}^{\text{sel}}_{<}(S)$ the set of clauses C such that $S' \rightarrow^{\sigma}_{\text{sel},<} C$ where all the clauses in S' are pairwise variable-disjoint renamings of clauses in S.
For technical convenience, we adopt the convention in the definition of $S \rightarrow^{\sigma}_{\text{sel},<} C$ that S contains only the premises of the rule and that the clauses in S are *not* renamed (they are treated as rigid variables). This implies that, for instance, we may construct the derivation $\{f(x) \simeq g(y), f(x) \simeq h(y)\} \rightarrow^{id}_{\text{sel},<} (h(y) \simeq g(y))$ but **not** $\{f(x) \simeq g(y), f(x) \simeq h(y)\} \rightarrow^{id}_{\text{sel},<} (h(y) \simeq g(y'))$, because the two occurrences of y are *not* renamed. Similarly, $\{f(x) \simeq a, f(g(x)) \simeq b\} \not\rightarrow^{\sigma}_{\text{sel},<} (a \simeq b)$, for any substitution σ because x and $g(x)$ cannot be unified. This does not destroy refutational completeness because the clauses are always renamed before applying $\rightarrow^{\sigma}_{\text{sel},<}$. In some sense, this version of the superposition calculus separates the renaming step from the application of the inference rules.

A *derivation* of a clause C_n from a set of clauses S (w.r.t. an ordering $<$ and a selection function sel) is a sequence of clauses $C_1, \ldots, C_n$ such that for every $i \in [1..n]$, $C_i \in S \cup \mathcal{S}^{\text{sel}}_{<}(\{C_1, \ldots, C_{i-1}\})$. A *refutation* is a derivation $C_1, \ldots, C_n$ such that $C_n = \Box$.

A set of clauses S is *saturated* if every clause $C \in \mathcal{S}^{\text{sel}}_{<}(S)$ is redundant in S. If S is saturated and if for every clause $C \in S$, $\text{sel}(C)$ either contains a negative literal, or contains all the literals $L \in C$ such that there exists a ground substitution σ such that $L\sigma$ is maximal in $C\sigma$, then S is satisfiable if $\Box \notin S$ (see [5] for details).

The relation $\equiv^{S}_{C}$

This relation relates some pairs of terms (or atoms, clauses, ...) (t, s) that have the same value in every interpretation in which S holds but not C. This is the case for instance when S contains a clause of the form $(t \simeq s) \vee C$. It is first defined on constant symbols, and then extended to other expressions by substitutivity.

Definition 2 Let S denote a set of clauses and C be a flat, ground clause. We denote by $\equiv_C^S$ the smallest reflexive relation on constant symbols such that $a \equiv_C^S b$ if there exists a clause $(a \simeq b) \vee D \in S$ where $D \subseteq C$.

This relation is extended to every expression (term, atoms or clause) as follows: $f(t_1, \ldots, t_n) \equiv_C^S g(s_1, \ldots, s_m)$ iff $f = g$, $n = m$ and for every $i \in [1..n]$, $t_i \equiv_C^S s_i$ (f and g denote either function symbols or logical symbols).

Example 1 Let $S = \{a \simeq b \vee c \not\simeq d, a \simeq c \vee e \simeq d\}$ and $C = c \not\simeq d \vee e \simeq d$. We have $a \equiv_C^S b$, $a \equiv_C^S c$ and $f(a, a) \equiv_C^S f(b, c)$. Notice that we do *not* have $b \equiv_C^S c$ (the relation $\equiv_C^S$ is reflexive and symmetric, but not transitive in general).

If $t \equiv_C^S s$, then t and s are identical up to a renaming of constant symbols, but the converse does not hold: the authorized renamings of constant symbols is restricted by S and C.

2.3 I_0-flat and I_{nv}-closed Terms

In this section we characterize two syntactic classes of terms and we prove some useful properties of these classes. The classes are defined by imposing that some of the arguments of function symbols are not variables, and that others are flat. Informally, $\mathrm{I}_0(f)$ denotes the set of indices of the arguments of f that must be flat and $\mathrm{I}_{\mathrm{nv}}(f)$ denotes the set of indices that must not be variables.

2.4 Definition

Definition 3 We associate to every function symbol f of arity n two sets of indices $\mathrm{I}_0(f)$ and $\mathrm{I}_{\mathrm{nv}}(f)$ in $[1..n]$ such that $\mathrm{I}_0(f) \cup \mathrm{I}_{\mathrm{nv}}(f) = [1..n]$. A term t is:

- I_0-*flat* if it is not a variable, and for every subterm $f(t_1, \ldots, t_n)$ of t and for every $i \in \mathrm{I}_0(f)$, $t_i \in T_0$.
- I_{nv}-*closed* if it is not a variable, and for every subterm $f(t_1, \ldots, t_n)$ of t and for every $i \in \mathrm{I}_{\mathrm{nv}}(f)$, $t_i \not\in \mathcal{V}$.

A set of clauses S is I_0-flat (resp. I_{nv}-closed) if all non-variable terms occurring in S are I_0-flat (resp. I_{nv}-closed).

Example 2 Let f be a function symbol of arity 3. Assume that $\mathrm{I}_0(f) = \{1, 2\}$ and that $\mathrm{I}_{\mathrm{nv}}(f) = \{2, 3\}$. Then:

- $f(x, a, b)$ and $f(a, b, f(a, b, c))$ are I_{nv}-closed and I_0-flat,
- $f(x, x, a)$ is I_0-flat but not I_{nv}-closed because of index 2,
- $f(f(a, b, c), a, b)$ is I_{nv}-closed but not I_0-flat because of index 1,
- $f(f(a, a, b), x, b)$ is neither I_0-flat nor I_{nv}-closed because of indices 1 and 2.

Remark 1 Since $\mathrm{I}_0(f) \cup \mathrm{I}_{\mathrm{nv}}(f) = [1..n]$ where n is the arity of f, there is a close relation and balance between I_0-flat and I_{nv}-closed terms. For example, we can easily ensure that every non-variable term is I_0-flat by taking $\mathrm{I}_0(f) = \emptyset$. But in this case we must have $\mathrm{I}_{\mathrm{nv}}(f) = [1..n]$ thus no such term is I_{nv}-closed, except if it is ground. Similarly, if $\mathrm{I}_{\mathrm{nv}}(f) = \emptyset$ then every non-variable term is I_{nv}-closed, but the only terms that are I_0-flat are those of depth 0 or 1. The actual trade-off between $\mathrm{I}_0(f)$ and $\mathrm{I}_{\mathrm{nv}}(f)$ is to be fixed according to the axioms that are handled.

As we shall see in the next sections, the terms that we consider in the paper are always I_0-flat. The terms considered in Section 4 are also I_{nv}-closed, but this is not the case in Sections 3 and 5 in which we use a refined condition which is less restrictive.

We prove that I_0-flatness and I_{nv}-closedness are preserved by flat substitutions and by replacements.

Lemma 1 *Let σ be a flat substitution. If t is a I_0-flat (resp. I_{nv}-closed) term then $t\sigma$ is I_0-flat (resp. I_{nv}-closed).*

Proof Since σ is flat, any complex subterm in $t\sigma$ is of the form $f(t_1, \ldots, t_n)\sigma$, where $f(t_1, \ldots, t_n)$ occurs in t. Moreover, we have obviously $t_i \in T_0 \Rightarrow t_i\sigma \in T_0$ (since σ is flat) and $t_i \notin \mathcal{V} \Rightarrow t_i\sigma \notin \mathcal{V}$. Thus the conditions of Definition 3 are preserved by flat substitutions.

Lemma 2 *Let t, s be two non-variable terms and let p be a non-variable position in t such that $t|_p \in \Sigma_0 \Rightarrow s \in \Sigma_0$. If t, s are I_0-flat (resp. I_{nv}-closed) then $t[s]_p$ is I_0-flat (resp. I_{nv}-closed).*

Proof We prove the result by induction on the length of p. If $p = \varepsilon$, then the result is obvious, since $t[s]_p = s$. Otherwise, $p = i.q$, which means that $t = f(t_1, \ldots, t_n)$ and $t[s]_p = f(t_1, \ldots, t_{i-1}, t_i[s]_q, t_{i+1}, \ldots, t_n)$. By definition of p, t_i cannot be a variable, thus since t is I_0-flat (resp. I_{nv}-closed), so is t_i; furthermore, $t|_p \in \Sigma_0 \Rightarrow t_i|_q \in \Sigma_0$. We can therefore apply the induction hypothesis: $t_i[s]_q$ is I_0-flat (resp. I_{nv}-closed), and so is $t[s]_p$.

The next lemma states that the m.g.u. of two I_0-flat and I_{nv}-closed terms is necessarily flat.

Lemma 3 *Let s and t be two I_0-flat and I_{nv}-closed terms that are unifiable, and let σ denote an m.g.u. of s and t. Then σ is flat.*

Proof By definition, neither s nor t can be a variable, and if they are both constants then the result is obvious. Now assume that $s = f(s_1, \ldots, s_n)$, and $t = f(t_1, \ldots, t_n)$, we prove the result by induction on the size of s.

Let $\theta_0 = \emptyset$, and for $i \in \{1, \ldots, n\}$, let μ_i be an m.g.u. of $\{s_i\theta_{i-1} =^? t_i\theta_{i-1}\}$ and $\theta_i = \theta_{i-1}\mu_i$. Obviously, θ_n is an m.g.u. of $\{s =^? t\}$. We prove that for all $i \in \{1, \ldots, n\}$, μ_i is flat. Since θ_0 is trivially flat, a simple induction using Proposition 1 allows to prove that every θ_i is also flat. Suppose θ_{i-1} is flat. Then by Lemma 1 $s\theta_{i-1}$ and $t\theta_{i-1}$ are both I_0-flat and I_{nv}-closed, and there are two cases to consider.

- If $s_i\theta_{i-1}$ is of the form $g(u_1, \ldots, u_m)$, where $m \geq 1$, then i cannot be in $\mathrm{I}_0(f)$ since $s\theta_{i-1}$ is I_0-flat. Thus $i \in \mathrm{I}_{\mathrm{nv}}(f)$, since $\mathrm{I}_0(f) \cup \mathrm{I}_{\mathrm{nv}}(f) = [1..n]$ (see Definition 3). Since $t\theta_{i-1}$ is I_{nv}-closed, this implies that $t_i\theta_{i-1}$ is not a variable, thus it is also of the form $g(v_1, \ldots, v_m)$. Since the size of $s_i\theta_{i-1}$ is strictly smaller than that of s, by the induction hypothesis, μ_i is I_0-flat.
- If $s_i\theta_{i-1}$ is a variable, then $i \in \mathrm{I}_0(f)$, since $s\theta_{i-1}$ is I_{nv}-closed. Since $t\theta_{i-1}$ is I_0-flat $t_i\theta_{i-1}$ is either a variable or a constant. Thus, $\mu_i = \{s_i\theta_{i-1} \mapsto t_i\theta_{i-1}\}$ is flat. The case where $t_i\theta_{i-1}$ is a variable is similar. Finally, if $t_i\theta_{i-1}$ and $s_i\theta_{i-1}$ are both constant symbols then $\mu_i = id$ and $\theta_i = \theta_{i-1}$.

3 An instantiation-based proof procedure

A close inspection of the instantiation phases of the superposition calculus in the rewrite-based approach to SMT problems revealed that for the considered background theories, it is sufficient to instantiate the axioms of the theory using only the ground

terms appearing in the original problem. The main idea we exploit in our instantiation scheme is the fact that the flattening operation permits to view constants as names for complex terms. This is why we focus on instantiations based only on constants.

Let S be a set of clauses whose satisfiability we wish to test by considering a finite set of its ground instances. A first, intuitive way to instantiate the non-ground clauses in S consists in replacing all variables by the constants appearing in S *in every possible way*. A formal definition of the resulting set follows.

Definition 4 Given a set of clauses S, we let $S_{\rm c}$ denote the set

$$S_{\rm c} = \{C\theta \mid C \in S,\ \forall x, x\theta \text{ is a constant in } S\}.$$

If $S_{\rm c}$ is unsatisfiable, then so is S; yet, it is clear that in general, $S_{\rm c}$ may be satisfiable although S is not. However, it is possible to determine a sufficient condition of the equisatisfiability of $S_{\rm c}$ and S, based on the inferences that derive the empty clause starting from S.

Definition 5 Given a set of clauses S, we denote by $\mathcal{SF}^{\rm sel}_{<}(S)$ the set of clauses $C \in \mathcal{S}^{\rm sel}_{<}(S)$ such that the mgu of the inference that generated C is flat.

A derivation $C_1, \dots, C_n$ is *flat* iff for every $i \in [1..n]$, $C_i \in S \cup \mathcal{SF}^{\rm sel}_{<}(\{C_1, \dots, C_{i-1}\})$.

The instantiation of the clauses with all the constants occurring in the original set of clauses is complete if the latter admits a flat refutation:

Theorem 1 *If S admits a flat refutation, then $S_{\rm c}$ is unsatisfiable.*

Proof Let $C_1, \dots, C_n$ denote a flat refutation of S. Let $S'_1, \dots, S'_n$ denote the sequence such that:

- $S'_1 = S_{\rm c}$;
- for all $i = 2..n$, $S'_i = S'_{i-1} \cup \mathcal{S}^{\rm sel}_{<}(S'_{i-1})$.

We prove by induction on k that for any flat ground substitution θ the clause $C_k\theta$ is in S'_k. Thus, since $C_n = \Box \in S_n$, then $\Box \in S'_n$, which proves that $S_{\rm c}$ is unsatisfiable.

If $k = 1$ then the result is obvious: C_1 is a clause in S, hence, by construction, $C\theta$ is a clause in $S_{\rm c}$. Now assume the result is true for some $k \geq 1$, consider a flat ground substitution θ. We assume that C_{k+1} is obtained by a paramodulation from $D_1 = u \simeq v \vee E_1$ into $D_2 = l[u'] \simeq r \vee E_2$ occurring in $C_1, \dots, C_k$; the proof in all the other cases is similar. Here, $C_{k+1} = (l[v] \simeq r \vee E_1 \vee E_2)\sigma$, and by hypothesis, σ is flat. Let $\mu = \sigma\theta$, then μ is also a flat ground substitution; therefore, by the induction hypothesis, both $D_1\mu$ and $D_2\mu$ are in S'_k (since the sequence (S'_k) is monotonic). Since the paramodulation of $D_1\mu$ into $D_2\mu$ generates $C_{k+1}\mu$, the latter is in S'_{k+1}, which concludes the proof.

The main inconvenience is that $S_{\rm c}$ can be a very large set. We thus define an instantiation scheme which generates a set of ground clauses potentially much smaller than $S_{\rm c}$, while still restricting ourselves to instantiations based on constants.

3.1 Definition of the instantiation scheme

The basic idea of our instantiation scheme is close to the one of existing instantiation-based methods (see for instance [28, 21]): namely, to use unification in order to generate relevant instances. For example, from the clauses $f(x) \simeq a, f(b) \not\simeq a$, we shall generate, by unifying the terms $f(x)$ and $f(b)$, the instance $f(b) \simeq a$ of the first clause, yielding the unsatisfiable ground set $\{f(b) \simeq a, f(b) \not\simeq a\}$. More generally, if C is a clause containing a (non variable) term t and if s is a term occurring in the clause set, we shall derive the instance $C\sigma$ such that σ is a unifier of t and s. This naive instantiation procedure has two drawbacks: first it is not complete in the sense that the obtained set of instances may be satisfiable even if the initial clause set is unsatisfiable and has a flat refutation (e.g. $f(x, a) \simeq a, f(c, b) \not\simeq a, a \simeq b\}$). Second, it does not terminate in general, as exhibited by the set $\{\neg p(x) \vee p(f(x))\}$ from which the infinite set of instances $\{\neg p(f^i(x)) \vee p(f^{i+1}(x))\}$ can be generated. Thus we need to refine it. This is done as follows.

First we restrict ourselves to *flat* unifiers, which guarantees that the instantiation scheme always terminates. This is possible since the clause sets we consider will always admit flat refutations. Second, we use a *relaxed* (pseudo-)unification algorithm which only compares the head symbol of the terms. More precisely, two (proper) subterms of t, s are always taken to be identical, except if one of them is a variable x and the other is a constant symbol c, in which case we add the binding $x \mapsto c$ into the resulting substitution σ. If two such bindings occur, namely $x \mapsto a, x \mapsto b$, where $a \neq b$, then the unification algorithm still succeeds (a and b are taken to be identical), and by convention, the smallest constant symbol among a, b according to the ordering $<$ is chosen as the value of x. Bindings of the form $x \mapsto f(t_1, \dots, t_n)$ where $n > 0$ are simply ignored: the unification succeeds and an empty pseudo-unifier is returned.

The underlying idea is that we do not want to compute only the unifiers of t and s, but rather all the *flat* unifiers of the terms t', s' that can (at least potentially) be obtained from t and s after some superposition steps below the root. For instance the clauses $f(a, x) \simeq a$ and $f(c, b) \simeq c$ would yield the instance $f(a, b) \simeq a$, although the terms $f(a, x)$ and $f(c, b)$ are not unifiable, because they may become so later in the derivation, e.g. if a is replaced by c by superposition. Similarly the clauses $f(x, x) \simeq a$ and $f(b, c) \simeq c$ should yield the instance $f(b, b) \simeq a$ (or $f(c, c) \simeq a$ but not both). On the other hand $f(x)$ and $g(y)$ are still not unifiable, but $f(f(x))$ and $f(g(y))$ are, with an empty unifier, since they can be reduced to the same term by superposition below the root, e.g. using $f(a) \simeq a, g(b) \simeq a$. Similarly $f(x, g(x))$ and $f(g(y), y)$ have an empty pseudo-unifier.

The reason for this rather unusual decision is that we cannot use the superposition calculus to compute an "exact" set of substitutions (as is done in [21]). Indeed, we wish to devise an always-terminating instantiation procedure and the superposition calculus does not terminate in general on the classes we consider. We need to reason only on the set of terms that already occur in the original clause set, using an over-approximation of the set of flat unifiers, which as we shall see is sufficient in our context.

The fact that clauses are only instantiated with flat substitutions is not such a limitation since in practice, fresh constants that serve as names for ground terms can be introduced during the flattening operation. Thus, in a sense, the only limitation of this scheme is that instantiations only involve ground terms occurring in the original set of clauses.

Definition 6 Let $t = f(t_1, \dots, t_n)$ and $s = f(s_1, \dots, s_n)$ be two terms. We denote by $\sim_{(t,s)}$ the smallest equivalence relation on T_0 for which $t_i \sim_{(t,s)} s_i$, for all $i \in [1..n]$ such that $t_i, s_i \in T_0$.

The *pseudo-unifier* of two terms t, s with the same head symbols is the substitution σ defined as follows: $dom(\sigma) \stackrel{\text{def}}{=} \{x \in \mathcal{V} \mid \exists c \in \Sigma_0, x \sim_{(t,s)} c\}$, and for all $x \in dom(\sigma)$,

$$x\sigma \stackrel{\text{def}}{=} \min_{<}\{c \mid c \in \Sigma_0, x \sim_{(t,s)} c\}.$$

The *agreement condition* of two terms $f(t_1, \dots, t_n)$ and $f(s_1, \dots, s_n)$ is the disjunction of the disequations $c \not\simeq d$, where c and d are distinct constants such that $c \sim_{(t,s)} d$.

Example 3 Consider the terms $t = f(g(x), a, x, b, x, z, z')$ and $s = f(h(y), y, c, d, b, z', z)$ and the ordering $a \prec b \prec c \prec d \prec x \prec y \prec z' \prec z$. Then there are three equivalence classes: $a \sim_{(t,s)} y, x \sim_{(t,s)} b \sim_{(t,s)} c \sim_{(t,s)} d$, and $z \sim_{(t,s)} z'$. Thus, the pseudo-unifier of t and s is $\{x \mapsto b, y \mapsto a\}$. The agreement condition of t and s is $b \not\simeq c \vee b \not\simeq d \vee c \not\simeq d$.

Intuitively, if t and s are two terms of depth 1, then their agreement condition encodes equality conditions on constants that make these terms unifiable.

The relation $\sim_{(t,s)}$ is preserved by a particular class of substitutions:

Proposition 4 *Let t, s be two terms and σ be a substitution that maps every variable in its domain to another variable. If $u \sim_{(t,s)} v$ then $u\sigma \sim_{(t\sigma,s\sigma)} v\sigma$.*

Definition 7 The **Instantiation Rule (I)** is defined as follows:

$$S \to S \cup \{C\sigma \vee E\}$$

if the following conditions hold:

- C is a clause in S.
- D is a renaming of a clause in S (possibly C), sharing no variable with C.
- σ is the pseudo-unifier of two terms $t = f(\mathbf{t})$ and $s = f(\mathbf{s})$ occurring in C, D respectively.
- E is the agreement condition of t and s.

We denote by $\hat{S}$ the set of clauses that can be generated from S using the instantiation rule above. By definition, every clause in $\hat{S}$ is subsumed by an instance of a clause in S.

Example 4 Let S denote the set of clauses containing the clauses

$$\begin{array}{llllll} 1: & \mathtt{cons}(\mathtt{car}(x), \mathtt{cdr}(x)) \simeq x, & 2: & \mathtt{car}(\mathtt{cons}(x,y)) \simeq x, & 3: & \mathtt{cdr}(\mathtt{cons}(x,y)) \simeq y, \\ 4: & \mathtt{car}(a) \simeq b, & 5: & \mathtt{cons}(a,c) \simeq d, & 6: & \mathtt{car}(\mathtt{cdr}(b)) \simeq c. \end{array}$$

The clauses other than those in S that are generated by the instantiation scheme are represented in Figure 2. The first column of the table contains the complex terms that are considered for the pseudo-unification, the second column contains the numbers of the clauses in which these terms occur, and the third column contains the instantiated clause. Note that the pairs of terms that are considered do not generate any agreement condition.

Complex terms	Involved clauses	Instantiated clause
$\texttt{car}(x), \texttt{car}(a)$	(1), (4)	$\texttt{cons}(\texttt{car}(a), \texttt{cdr}(a)) \simeq a$
$\texttt{cdr}(x), \texttt{cdr}(b)$	(1), (6)	$\texttt{cons}(\texttt{car}(b), \texttt{cdr}(b)) \simeq b$
$\texttt{cons}(x, y), \texttt{cons}(a, c)$	(2), (5)	$\texttt{car}(\texttt{cons}(a, c)) \simeq a$
$\texttt{cons}(x, y), \texttt{cons}(a, c)$	(3), (5)	$\texttt{cdr}(\texttt{cons}(a, c)) \simeq c$

Fig. 2 Instantiated clauses of Example 4.

The rule (I) should be compared with existing instantiation schemes such as the one in [23]. In [23], sets of instances are computed by assigning a set of ground terms to each variable and to each argument of function symbols. Set constraints are then generated to encode the relations between these sets of instances (propagating unification conditions between the terms) and a minimal solution of these constraints is constructed. The instantiation scheme of [23] is more general in some respect than ours because non-flat instantiation can be constructed. But termination is not ensured, except for some particular classes of formulae for which the depth of the non-interpreted terms is bounded, such as the one of *stratified clause sets* (this class can be also captured by our approach, see Section 6.2). However, the approach in [23] generates much more instances than the rule (I) (thus may be less efficient) because each variable is instantiated independently from the others. Consider for instance a formula containing the terms $f(x_1, \ldots, x_n)$ and $f(a_1, \ldots, a_1), \ldots, f(a_k, \ldots, a_k)$. Applying the instantiation scheme of [23] yields to consider the following set constraints: $\{a_1, \ldots, a_k\} \in A_{f,i}$ (meaning that each constant a_j for $j \in [1..k]$ may occur as the i-th argument of f). Since the variable x_i also occurs as the i-th argument of f, this entails that x_i must be instantiated by $a_1, \ldots, a_k$. But then we get k^n possible instances. In contrast, our procedure generates only k instances $f(a_1, \ldots, a_1), \ldots, f(a_k, \ldots, a_k)$. On the other hand, [23] performs instantiation modulo arithmetic, which is not tackled by the approach described in the present paper (instantiation modulo integers is considered in [20]).

Our technique is closer from the one originated by [28], except that non-flat instances are simply ignored.

In general the set $\hat{S}$ is not ground, since it contains S. We assume that Σ contains a special constant symbol λ, not occurring in S; all the variables occurring in $\hat{S}$ are instantiated with this constant.

Definition 8 Given a set of clauses S, we denote by S_λ the set of clauses obtained by replacing all variables in S by λ.

We provide another example of an application of the instantiation scheme.

Example 5 We consider the following clause set S:

$$
\begin{array}{ll}
1 & p(a, x) \simeq \texttt{true} \vee q(x, x) \not\simeq \texttt{true} \\
2 & p(c, d) \not\simeq \texttt{true} \\
3 & q(f(x), x) \simeq \texttt{true} \\
4 & a \simeq b \\
5 & b \simeq c \\
6 & f(x) \simeq x
\end{array}
$$

The instantiation rule generates the following clauses. We specify for each clause the clause from which it is obtained and the considered term.

7	$p(a,d) \simeq \mathtt{true} \vee q(d,d) \not\simeq \mathtt{true} \vee a \not\simeq c$	1, $p(c,d)$
8	$q(f(d),d) \simeq \mathtt{true}$	3, $q(d,d)$
9	$f(d) \simeq d$	6, $f(d)$

It is simple to verify that the set of ground clauses $\{2,4,5,7,8,9\}$ is unsatisfiable; hence so is $\hat{S}_\lambda$, which contains clauses 1 through 9 where all variables are instantiated with λ. Thus it happens that the instantiation scheme is complete on this clause set. Section 3.2 will show that it is not always the case and provide tractable conditions ensuring completeness.

Getting rid of Agreement Conditions

It is clear from the definition that the clauses in $\hat{S}$ are obtained from instances of S by adding flat ground clauses. The addition of these flat ground clauses is actually not necessary: all the results in this paper still hold if they are not added at all. Indeed, if $\hat{S}'$ denotes the set of clauses obtained from $\hat{S}$ by removing agreement conditions, then soundness is obvious since $\hat{S}'$ only contains instances of clauses in S, so that $S \models \hat{S}'$, and (partial) completeness follows from the fact that the clauses in $\hat{S}$ are logical consequences of $\hat{S}'$. Since different agreement conditions can be added to the same instance of a clause in S, the set $\hat{S}$ may be significantly larger than $\hat{S}'$. We nevertheless prefer to add these conditions explicitly for two reasons. Firstly this makes the clause sets more general hence significantly strengthens the completeness result, which is of some theoretical interest. Secondly, from a practical point of view, $\hat{S}'$ may actually be easier to refute than $\hat{S}$ by an SMT-solver, since agreement conditions may enable to identify and discard useless clauses. For instance if $\hat{S}'$ contains a clause $C \vee (a \not\simeq b)$, where $a \not\simeq b$ is a agreement condition, and if the system is considering a partial model in which $a \not\simeq b$ holds, then the clause C can simply be discarded, since it is subsumed. If the agreement condition is dropped, then C must be considered although it is actually useless. Of course, practical experimentations are needed to investigate the effect of the removal/addition of these conditions, and one may probably have to find a trade-off depending of the number of clauses that must be added.

A good compromise would be to *remove* these agreement conditions, in order to keep the size of the clause set as low as possible, but for every disequation $a \not\simeq b$ occurring in the agreement condition, to replace any occurrence of a by b (or conversely), provided that this occurrence was introduced by instantiation using a pseudo-unifier (i.e. that constant a does not occur in the initial clause). This technique ensures that soundness and (partial) completeness are preserved since the obtained clause set is still less general than S but more general than $\hat{S}$. Constants already occurring in the initial clause should *not* be replaced, otherwise soundness would be lost.

This compromise is used in the complexity results of the following section and in the experiments.

Complexity Results

If C is a clause in S containing n distinct variables and there are m constant symbols occurring in S, then there are at most m^n pseudo-unifiers that can be applied to C. This entails the following result:

Theorem 2 *Given a set of clauses S, let n denote the maximal number of distinct variables appearing in a clause in S, and m denote the number of constants occurring in S. Then the maximal number of clauses that can be generated by the instantiation rule (without agreement conditions) is $O(|S|m^n)$.*

This result is important from a complexity point of view, especially when considering $\mathcal{T}$-satisfiability problems (i.e. testing the satisfiability of a conjunction of ground literals modulo a theory $\mathcal{T}$). Indeed, if theory $\mathcal{T}$ is fixed and it is guaranteed that the instantiation scheme is correct for any set of clauses of the form $\mathcal{T} \cup S$, where S is a set of ground unit clauses, then the number of distinct variables appearing in a clause is a constant, which means that a *polynomial* set of ground clauses is generated. If $\mathcal{T}$ is Horn, then the generated set of ground clauses is also Horn, and its satisfiability can be tested in polynomial time. This result shows an advantage of the instantiation scheme compared to rewrite-based approaches: this scheme avoids the generation of exponentially many clauses in, e.g., the theory of arrays [3], contrary to the original rewrite-based approach [2], or the more recent decompositional approach [9]. For the theory of records with extensionality [2], the instantiation scheme generates a polynomial set of ground Horn clauses, whose satisfiability can be tested in polynomial time. We thus obtain a polynomial satisfiability procedure for this theory with no effort, a result that had already been obtained for this specific theory in [8]. Note that the original rewrite-based approach is an exponential satisfiability procedure for the theory of records with extensionality [2].

3.2 Completeness of the instantiation scheme

Since the set $\hat{S}$ obtained by our instantiation scheme only contains flat instances of S along with agreement conditions, it is obviously possible for S to be unsatisfiable whereas $\hat{S}$ is satisfiable. For example, this is the case if no refutation for S is flat. Even if S admits a flat refutation, $\hat{S}_\lambda$ may be satisfiable, as evidenced by the set of clauses $S = \{f(g(x)) \simeq a, g(y) \simeq b, f(z) \not\simeq a\}$. Here, $\hat{S} = S$, and $\hat{S}_\lambda = \{f(g(\lambda)) \simeq a, g(\lambda) \simeq b, f(\lambda) \not\simeq a\}$ is satisfiable. But S admits a flat refutation:

$$\begin{array}{ll} f(b) \simeq a & \text{(superposition, } y \mapsto x) \\ a \not\simeq a & \text{(paramodulation, } z \mapsto b) \\ \square & \text{(reflexivity)} \end{array}$$

Such a behaviour may occur even if S contains no function symbol, as evidenced by the set of clauses $S = \{x \not\simeq a, b \simeq a\}$.

We introduce some semantic conditions on a set of clauses S which ensure that the instantiation scheme is complete, i.e. that S and $\hat{S}_\lambda$ are equisatisfiable. This result is a first step towards the definition of syntactic conditions that ensure completeness.

A semantic criterion ensuring completeness

Roughly speaking, the instantiation scheme is complete when S admits a derivation satisfying particular properties. Firstly, all the unifiers occurring in the derivation must be flat and all the clauses must be non-variable-eligible (see Definition 1). Furthermore, no superposition step replacing a term by a variable can occur in the derivation. Finally all the clauses must be I_0-flat (see Definition 3), and the inferences involving clauses that are not I_{nv}-closed must be strongly restricted: either the non-I_{nv}-closed premise occurs in the initial clause set, or the inference is ground (with an empty unifier).

Definition 9 (Simple Derivation) Let S be a set of clauses. A derivation δ of S is *simple* if it satisfies the following conditions:

1. All the unifiers occurring in δ are flat.
2. All the clauses occurring in δ are I_0-flat and non-variable-eligible.

3. If two terms t, s are unified in δ with mgu σ and if there is a position p such that $t|_p = f(t_1, \ldots, t_m)$ (resp. $s|_p = f(t_1, \ldots, t_m)$) and $t_j \in \mathcal{V}$ for some $j \in \mathrm{I_{nv}}(f)$, then either $s|_p$ (resp. $t|_p$) occurs in S or $\sigma = id$.
4. If an application of the superposition/paramodulation rule replaces a term $u\sigma$ by $v\sigma$, then $v\sigma \notin \mathcal{V}$.

A set of clauses S is *simply provable* if for all clauses C, if there exists a derivation of C from S, then there also exists a simple derivation of C from S. A class of clause sets $\mathfrak{S}$ is *simply provable* if every set of clauses in $\mathfrak{S}$ is simply provable.

Condition 4 forbids inferences such as $f(a) \simeq b, a \simeq x \vdash f(x) \simeq b$ (replacement of a term by a variable). Condition 3 states that the subterms that are not $\mathrm{I_{nv}}$-closed can only be unified with terms already occurring in the initial clause set. Notice that Conditions 2 and 3 are strongly related. It is easy to guarantee that all the terms in the derivation are $\mathrm{I_0}$-flat by taking $\mathrm{I_0}(f) = \emptyset$ for every $f \in \Sigma$ but then we must have $\mathrm{I_{nv}}(f) = [1..n]$, where n is the arity of f, thus Condition 3 cannot hold, unless the premises of every non-ground inference step are in S. Conversely, Condition 3 always holds if $\mathrm{I_{nv}}(f) = \emptyset$ but then by Condition 2 all the terms must be of depth 1 (see also Remark 1).

By Condition 1, every simple derivation is flat, but the converse does not hold. Condition 4 prevents replacement of a term by a variable which guarantees that inferences preserve Condition 2. Condition 3 is the less natural one. Intuitively it ensures that a variable is never unified with a constant symbol that has previously replaced a complex term by superposition. This allows us to discard derivations such as the one in Figure 3. In Figure 3, the term $f(x_2)$ is unified with $f(b)$ but since the latter does not occur in the original set of clauses, we cannot compute the substitution $x \mapsto b$ using the Instantiation rule only. For this purpose, we ensure that the subterms that are not $\mathrm{I_{nv}}$-closed ($f(x_2)$ in this example) are unified only with terms occurring in the original set of clauses. Notice that if we assume that every term occurring in the derivation is $\mathrm{I_{nv}}$-closed, then Condition 4 always holds, because by definition of $\mathrm{I_{nv}}$-closed terms, t_j cannot be a variable if $j \in \mathrm{I_{nv}}(f)$. However, our condition is less restrictive, and this will be useful in Section 5.

The following definition imposes an additional restriction on the unifiers occurring in a derivation:

Definition 10 (Pure Derivation) A substitution σ is *pure* if for every variable x, $x\sigma \in \mathcal{V}$. A derivation δ is *pure* if it is simple and if every unifier occurring in δ is pure.

For instance, the inference $f(x) \simeq a, f(y) \simeq b \vdash a \simeq b$ is pure, whereas $f(x) \simeq a, f(c) \simeq b \vdash a \simeq b$ is not (the unifier is $x \mapsto y$ in the first case and $x \mapsto c$ is the second one).

The following key result relates the two previous notions and will ensure completeness of the instantiation scheme.

Theorem 3 *If δ is a simple derivation of a clause C from a set of clauses S, then there exists a pure derivation of C from $\hat{S}$.*

Proof (Sketch) This result may seem surprising because pure derivations are strongly restricted. However, one can prove that all the instantiations that take place in the derivation δ can actually already be applied on the initial clause set, using only the

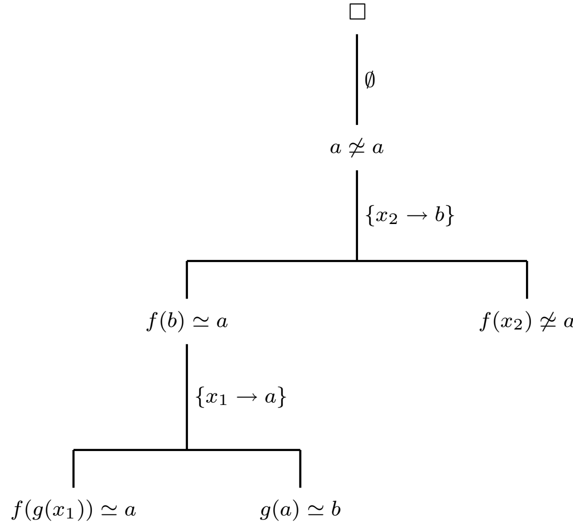

Fig. 3 A non-simple refutation.

instantiation rule of Definition 7. Assume that a variable x is instantiated by a non-variable term t at some point in the derivation. Then since δ is simple, t must be a constant symbol. Since the clauses occurring in the derivation are non-variable-eligible, x and t must occur in two terms $f(\dots, x, \dots)$ and $f(\dots, t, \dots)$ respectively, and the mapping $x \mapsto t$ is generated by unification. These two terms are necessarily obtained from two terms $f(\dots, x, \dots)$ and $f(\dots, s, \dots)$ occurring in parent clauses in the *initial* clause set (s may be distinct from t since the superposition rule can be applied on s, and x may actually be renamed). Moreover, by Condition 3 in Definition 9 the index of x in the term $f(\dots, x, \dots)$ cannot be in $\mathrm{I}_{\mathrm{nv}}(f)$, thus s cannot be a complex term (if it were, since $f(\dots, s, \dots)$ is I_0-flat, the index of the term s would not be in $\mathrm{I}_0(f)$, thus $f(\dots, x, \dots)$ would not be I_{nv}-closed, which contradicts our hypothesis). Thus the instantiation rule of Definition 7 can be applied to replace x by s.

The detailed proof is actually much more complex and highly technical since one has to handle properly all the necessary transformations on the derivation δ, in particular the swapping of variables and the replacement of constant symbols. We provide a very detailed proof in Appendix A.2.

We may therefore prove the completeness of the instantiation scheme for the class of simply provable sets of clauses:

Corollary 1 *If S is a simply provable and unsatisfiable set of clauses then $\hat{S}_\lambda$ is unsatisfiable.*

Proof This is a simple consequence of Theorem 3: since S is simply provable and unsatisfiable, it admits a simple derivation of □. By Theorem 3, $\hat{S}$ admits a pure derivation of □, and by instantiating all the variables in this derivation by λ, we obtain a refutation of the set $\hat{S}_\lambda$ which is therefore unsatisfiable.

Although we have proved the completeness of the scheme for the class of simply provable sets of clauses, there remains the issue of being able to detect such sets of

clauses, since this condition is semantic. The following two sections are devoted to the description of syntactic conditions ensuring that a clause set is simply provable.

4 A syntactic characterization of simply provable clause sets

In order to determine syntactic conditions that are general enough to be satisfied by several theories of interest, the potential inferences that can take place in a derivation need to be restrained. This is done by considering a particular selection function that satisfies some additional properties which guarantee that it is sufficiently restrictive. The idea is that a literal must be selected only if one of its ground instances is selected and that nonmaximal positive literals cannot be selected (the corresponding inferences are obviously useless). This selection function will be used throughout this section and the following ones.

Definition 11 We consider a selection function sel such that:

1. For every *ground* clause C, $\text{sel}(C)$ contains a unique literal L, and if L is positive then L is maximal in C.
2. For every non-ground clause C and every literal L in C, $L \in \text{sel}(C)$ if and only if there exists a ground substitution σ of the variables in C such that $L\sigma \in \text{sel}(C\sigma)$[3].

These conditions are strong, but such a selection function always exists. Condition 1 imposes some constraints on the selection of literals in ground clauses (this condition is easy to fulfill, for instance by selecting the maximal literal). Condition 2 actually defines the selection function on non-ground clauses, assuming that the function is defined for ground clauses. Note that the selection function is not intended to be employed in practice: it will be used only to show the *existence* of simple derivations.

4.1 Variable-Preserving clause sets

According to Definition 9 the first step towards determining conditions that guarantee the equisatisfiability of S and $\hat{S}_\lambda$ is ensuring that no derivation starting from S can generate a clause that is variable-eligible. Note that it is *not* sufficient to check that the initial clause set contains no variable-eligible clause, and that it is necessary to ensure that no variable-eligible clause can appear during proof search. An easy solution would be to discard all clauses containing variables at the root level, but this is much too restrictive (the theory of arrays for instance contains such clauses). Thus we first define a syntactic class of clauses, the class of *variable-preserving* clauses, for which the inference rules of the superposition calculus are guaranteed never to generate a variable-eligible clause. Other approaches ensuring the absence of variable-eligible clauses in derivations include [2,27,29]. Our approach consists in defining syntactic conditions that are tested on the original set of clauses, to guarantee the required property. This is done by defining variables that are I_{vp}*-constrained* in a clause, a condition that depends on the positions of their occurrences in the clause. Intuitively, no matter the inference, an I_{vp}-constrained variable cannot become eligible in a generated clause, because it always occurs as a strict subterm of a maximal literal in the clause.

Definition 12 We associate to each function symbol f of arity n a set of indices $\mathrm{I}_{vp}(f)$ in $[1..n]$. A variable x is I_{vp}*-constrained* in a term t if t contains a subterm of the form $f(t_1, \ldots, t_n)$ such that $t_i = x$ and $i \in \mathrm{I}_{vp}(f)$. A variable x is I_{vp}*-constrained in a literal* L if L is of the form $t \simeq s$ or $t \not\simeq s$ and x is I_{vp}-constrained in t or in s. A variable

[3] Note that $\text{sel}(C)$ may contain several literals.

is I_{vp}-*constrained in a clause* C if it is I_{vp}-constrained in at least one literal in C. If e is an expression (term, literal, or clause), we denote by $I_{\mathcal{V}}(e)$ the set of variables that are I_{vp}-constrained in e.

Example 6 Let $\Sigma = \{f, g\}$, where f is of arity 2 and g of arity 1, and suppose that $\mathrm{I}_{vp}(f) = \mathrm{I}_{vp}(g) = \{1\}$. If $C = x \simeq y \vee f(g(x), y) \simeq f(g(x), x)$, then x is I_{vp}-constrained in C and y is not.

In particular, if for all $n \in \mathbb{N}$ and for all $f \in \Sigma_n$ we have $\mathrm{I}_{vp}(f) = [1..n]$, and if e is a nonvariable expression, then $I_{\mathcal{V}}(e)$ is the set of variables in e. If $\mathrm{I}_{vp}(f) = \emptyset$ for all $f \in \Sigma$, then $I_{\mathcal{V}}(e) = \emptyset$. The sets of variable-preserving literals and clauses are defined by imposing constraints that ensure I_{vp}-constrained variables remain so.

Definition 13 A literal L is *variable-preserving* in a clause $C = L \vee D$ if:

1. L is a negative literal $t \not\simeq s$, and one of the following conditions is satisfied:
   (a) $s \in I_{\mathcal{V}}(t)$ or $t \in I_{\mathcal{V}}(s)$;
   (b) $t, s \notin \mathcal{V}$ and either $I_{\mathcal{V}}(t)$ or $I_{\mathcal{V}}(s)$ is empty;
   (c) $t, s \notin \mathcal{V}$ and $I_{\mathcal{V}}(t) \cup I_{\mathcal{V}}(s) \subseteq I_{\mathcal{V}}(D)$;
2. L is a positive literal $t \simeq s$, and one of the following conditions is satisfied:
   (a) $t, s \notin \mathcal{V}$ and $I_{\mathcal{V}}(t) = I_{\mathcal{V}}(s)$;
   (b) $\{t, s\} \subseteq T_0$ and $\{t, s\} \cap \mathcal{V} \subseteq I_{\mathcal{V}}(D)$.

A clause C is *variable-preserving* if every literal $L \in C$ is variable-preserving in C, and a set of clauses S is *variable-preserving* if every clause in S is variable-preserving.

In particular, equations of the form $id(x) \simeq x$ are forbidden. Just as in the case of I_0-flat and I_{nv}-closed terms, there is a trade-off between Conditions 2a-1b and 1a-2b. The former are easy to fulfill by assuming that $\mathrm{I}_{vp}(f)$ is empty for every f, but in this case the latter never hold.

Intuitively, Condition 2a ensures that the set of I_{vp}-constrained variables will be preserved during superposition: if x is I_{vp}-constrained in a term t and if s is obtained from t by superposition, then either x is instantiated or x is I_{vp}-constrained in s. Then a literal of the form $x \simeq t$ (resp. $x \not\simeq t$), where $x \in \mathcal{V}$, is allowed only if x is not eligible in C, or in the *descendants* of C, unless x is instantiated by unification. This is ensured either by checking that x occurs in t (Condition 1a) or that their always exists a literal in D that is greater than $x \simeq t$ (Condition 2b). These conditions will be preserved during the derivation because the set of I_{vp}-constrained variables is preserved.

Example 7 Let $C = f(x, y) \simeq f(a, y)$, $D = v \simeq a \vee f(u, v) \not\simeq b$. Assume that $\mathrm{I}_{vp}(f) = \{2\}$. The reader can verify that $I_{\mathcal{V}}(f(x, y)) = I_{\mathcal{V}}(f(a, y)) = \{y\}$ and $I_{\mathcal{V}}(f(u, v)) = \{v\}$. Thus C, D are both variable-preserving. The clause $E = u \simeq a \vee f(u, v) \not\simeq b$ is not variable-preserving, because $u \notin I_{\mathcal{V}}(f(u, v))$. Notice that if we take $\mathrm{I}_{vp}(f) = \{1, 2\}$ then we have $I_{\mathcal{V}}(f(u, v)) = \{u, v\}$ thus E is variable-preserving, but then C is not variable-preserving because $I_{\mathcal{V}}(f(a, y)) = \{y\} \neq I_{\mathcal{V}}(f(x, y)) = \{x, y\}$, and Condition 2a does not hold. The application of the superposition rule between C and E generates the clause $u \simeq a \vee f(a, v) \not\simeq b$ that is variable-eligible, thus not variable-preserving. This illustrates the importance of Condition 2a.

The following proposition is an immediate consequence of the definition.

Proposition 5 *Let C, D be two variable-preserving clauses. Then $C \vee D$ is variable-preserving.*

The main property of interest satisfied by variable-preserving clauses are that these clauses cannot be variable-eligible.

Proposition 6 *All variable-preserving clauses are non-variable-eligible.*

Proof Let C be a variable-preserving clause. First assume that C is of the form $x \not\simeq t \vee D$ where x is a variable. By Definition 13, $x \not\simeq t$ must satisfy Condition 1a, thus necessarily $x < t$. Now, assume that C is of the form $x \simeq t \vee D$, where x is a variable such that $x \not< t$, and assume that $x \simeq t$ is an eligible literal. Then, by Definition 13, one of Conditions (2a) or (2b) must hold. However, x is a variable, hence Condition (2a) cannot hold. Thus, Condition (2b) holds, $t \in T_0$ and $x \in I_{\mathcal{V}}(D)$. Furthermore, if t is a variable, then $t \in I_{\mathcal{V}}(D)$.

Since $x \simeq t \in \mathrm{sel}(D)$, there exists a ground substitution σ such that $\forall L \in D$, $x\sigma \simeq t\sigma \not< L\sigma$ by the conditions on the selection function (see Definition 11). Now since $x \in I_{\mathcal{V}}(D)$, there exists a term of the form $f(t_1, \ldots, t_n)$ occurring in D such that $t_i = x$. Thus, $x\sigma < f(t_1, \ldots, t_n)\sigma$ and necessarily, since $(x \simeq t)\sigma$ is maximal, $t\sigma \not< f(t_1, \ldots, t_n)\sigma$. This implies that $t\sigma$ cannot be a constant, and since $t \in T_0$, we must have $t \in \mathcal{V}$. But in this case, t must also be in $I_{\mathcal{V}}(D)$, which means that t occurs at a non-root position in a literal $L' \in D$. But then $(x \simeq t)\sigma < L'\sigma$, which is impossible.

An essential property of the class of variable-preserving clause sets, that is not satisfied by the class of non-variable-eligibleclauses, is that it is stable by superposition, if the associated unifier is flat:

Theorem 4 *Let S be a variable-preserving clause set. Assume that $S' \rightarrow^{\sigma}_{\mathrm{sel},<} C$, where all the clauses in S' are pairwise variable-disjoint renamings of clauses in S, and suppose that σ is flat. Then C is also variable-preserving.*

Proof (Sketch) A careful inspection of the different cases permits to show that the set of variable-preserving clauses is closed by disjunction, flat instantiation and by replacement of a term t by a non-variable term s such that $I_{\mathcal{V}}(t) = I_{\mathcal{V}}(s)$. Then the proof follows from the definition of the calculus. The detailed proof can be found in Appendix C. The proof is not difficult, but it is rather long because there are many cases to check.

4.2 Controlled sets of clauses

We now define the class of controlled sets of clauses. These sets do not generate variable-eligible clauses, and the additional constraints they satisfy intuitively ensure the mgus occurring in any derivation are flat. In order to ensure that the latter property holds, we shall use the results in Section 2.3: according to Lemma 3, it is sufficient to assume that all the terms occurring in the derivation are both I_0-flat and I_{nv}-closed.

Definition 14 A clause C is *controlled* if it is variable-preserving, I_0-flat and I_{nv}-closed. A set of clauses is *controlled* if all the clauses it contains are controlled.

Stability of controlled clauses

We provide conditions guaranteeing the stability of controlled clauses under certain conditions.

Lemma 4 *Let C be a variable-preserving clause and let σ be a flat substitution. Then $C\sigma$ is also variable-preserving.*

Proof See Appendix B

Lemma 5 *The following properties hold:*

1. *Every flat and ground clause is controlled.*
2. *If C and D are controlled, then so is $C \vee D$.*
3. *If σ is a flat substitution and C is a controlled clause, then so is $C\sigma$.*

Proof Item 1 is obvious, and item 2 is an immediate consequence of Proposition 5. For item 3, we verify that the conditions of Definition 14 are preserved:

- The clause $C\sigma$ remains variable-preserving by Lemma 4.
- By Lemma 1, I_0-flatness and I_{nv}-closedness are preserved by flat instantiations; thus, all controlled literals remain controlled after instantiation.

We also show that controlled clauses remain so after particular replacement operations, and that particular inference steps are guaranteed to generate such clauses.

Lemma 6 *Let C be a clause and let t, s be two non-variable terms. Assume that:*

- *s is I_0-flat and I_{nv}-closed;*
- *$I_{\mathcal{V}}(s) = I_{\mathcal{V}}(t)$;*
- *if $t \in \Sigma_0$, then $s \in \Sigma_0$.*

If $C[t]_p$ is controlled, then $C[s]_p$ is controlled.

Proof By Theorem 4, $C[s]_p$ is variable-preserving, and by Lemma 2, all its non-flat literals are I_0-flat and I_{nv}-closed (notice that p cannot be a variable position).

Lemma 7 *Let D_1, D_2 denote (not necessarily distinct) controlled clauses. If $\{D_1, D_2\} \rightarrow^{\sigma}_{\mathrm{sel},<} C$, then σ is flat and C is controlled.*

Proof Since D_1 and D_2 are variable-preserving, by Theorem 4, if σ is flat then C is variable-preserving. We distinguish three cases, according to the rule used to derive C.

C is generated by the superposition or paramodulation rule. Then D_1 is of the form $L[t]_p \vee D'_1$, D_2 of the form $u \simeq v \vee D'_2$, and C is of the form $(L[v]_p \vee D'_1 \vee D'_2)\sigma$, where $\sigma = mgu(t, u)$. By hypothesis, t and u are I_0-flat and I_{nv}-closed, and by Lemma 3, σ must be flat. By applying Lemma 5 (3), we deduce that $(L[t]_p \vee D'_1)\sigma$ and $(u \simeq v \vee D'_2)\sigma$ are both controlled. Since D_2 is variable-preserving, one of the conditions of Definition 13 must hold. If Condition 2a holds, then v cannot be a variable. If Condition 2b holds, then v cannot be a variable either, since $u \simeq v$ is selected. Furthermore, if u is a constant, the head symbol of v cannot be a function symbol, since otherwise, we would have $v > u$. Therefore, we have $u \in \Sigma_0 \Rightarrow v \in \Sigma_0$, and by Lemma 6, $(L[v]_p \vee D'_1)\sigma$ is controlled. By Theorem 4, $C = (L[v]_p \vee D'_1 \vee D'_2)\sigma$ is variable-preserving, and it is simple to verify that C satisfies the I_0-flatness and I_{nv}-closedness conditions.

C is generated by the reflection rule. This means that C is of the form $D'_1\sigma$, where $D_1 = (t \not\simeq s) \vee D'_1$ and $\sigma = mgu(t, s)$. If $t \simeq s$ is flat, then σ is obviously flat. Otherwise, since D_1 is controlled, t and s must be I_0-flat and I_{nv}-closed, thus, σ must be flat by Lemma 3. By Theorem 4, $C = D'_1\sigma$ is variable-preserving, and it is controlled by Lemma 5 (3).

C is generated by the equational factorisation rule. This means that C is of the form $(D_1' \vee s \not\simeq v \vee t \simeq s)\sigma$, where $D_1 = (D_1' \vee u \simeq v \vee t \simeq s)$, the selected literal is $t \simeq s$, and $\sigma = mgu(t,u)$. Since $t \simeq s$ is variable-preserving, one of Conditions 2a or 2b of Definition 13 must hold. First assume that Condition 2a holds, so that t is not a variable. If u is a variable, then it must occur in a complex term in $D_1' \vee t \simeq s$, which would prevent $t\sigma$ from being an eligible term. If u is a constant, then t must also be a constant, and $\sigma = id$ is flat. Otherwise, t and u must both be I_0-flat and I_{nv}-closed since D_1 is controlled, and again, σ must be flat by Lemma 3. Now assume that Condition 2b holds. Then t and s cannot be variables since otherwise the clause would be variable-eligible.This implies that u, v are flat (otherwise the literal $t \simeq s$ would not be selected). Then σ is flat. Therefore, C is variable-preserving by Theorem 4, and controlled by Lemma 5 (3).

Completeness

We obtain the main result of this section:

Theorem 5 *Every set of clauses that is controlled is simply provable.*

Proof We prove that all derivations for S are simple and only contain controlled clauses, by induction on their length. Let $\delta = C_1, \ldots, C_n$ be a derivation from S. By the induction hypothesis, the derivation $C_1, \ldots, C_{n-1}$ is simple and the clauses $C_1, \ldots, C_{n-1}$ are controlled. Then by Lemma 7, C_n is controlled, and we now verify that δ satisfies the conditions of Definition 9. Let σ be the unifier corresponding to the last inference of δ.

1. The clauses occurring in $\mathcal{T}$ are controlled, hence by Lemma 7, σ is flat.
2. Since C_n is controlled, it is variable-preserving, and by Proposition 6, it is not variable-eligible. Furthermore, every controlled clause is I_0-flat by definition.
3. Since C_n is controlled, every term it contains is I_{nv}-closed, which means that Condition 3 trivially holds.
4. Suppose C_n is generated by superposition/paramodulation of $D_1 = u \simeq v \vee D_1'$ into a clause D_2, with the term $u\sigma$ being replaced by the term $v\sigma$. Then by hypothesis, D_1 is variable-preserving, and one of Conditions (2a) or (2b) of Definition 13 must hold. If Condition (2a) holds, then neither v nor $v\sigma$ is a variable. If Condition (2b) holds, both u and v are flat, and if they are variables, then they must appear in $I_{\mathcal{V}}(D_1')$. Thus, the only way for $u \simeq v$ to be maximal in D_1 is for u and v to both be constants, hence $v\sigma$ cannot be a variable.

Therefore, δ is simple, which completes the proof.

If S is a controlled set of clauses that is unsatisfiable, then it is simply provable by Theorem 5. We deduce by Corollary 1 that $\hat{S}_\lambda$ is also unsatisfiable, which proves that our instantiation scheme is correct when applied to S.

5 $\mathcal{C}$-controllable clauses

Theorem 5 makes the class of controlled clause sets a good candidate for applying the instantiation method described in Section 3. However, several theories of interest are non-controlled. For instance, one of the axioms of the theory of arrays is (see, e.g., [3]):

$$\forall x, z, v, w.\ z \simeq w \vee \mathtt{select}(\mathtt{store}(x, z, v), w) \simeq \mathtt{select}(x, w),$$

and this axiom is not controlled. Indeed, either index 1 is in $\mathrm{I}_0(\mathtt{select})$ and in this case $\mathtt{select}(\mathtt{store}(x,z,v),w)$ is not I_0-flat, or $1 \in \mathrm{I}_{\mathrm{nv}}(\mathtt{select})$ and $\mathtt{select}(x,w)$ is not I_{nv}-closed.

The goal of this section is to overcome this problem by introducing a more general class that allows us to handle such theories. The idea is that non-controlled sets of clauses may be transformed into controlled ones using superposition from particular equations. For instance, in the above theory, by applying the superposition rule on the term $\mathtt{store}(x,z,v)$, we shall ensure that the variable x is instantiated by a constant symbol, so that $\mathtt{select}(x,w)\sigma$ becomes I_{nv}-closed.

5.1 Overview

We introduce the more general class of $\mathcal{C}$-*controllable clauses*, which is obtained by relaxing the variable-preservation and I_{nv}-closedness conditions on some of the terms that occur in the clauses. The terms for which the conditions can be relaxed are those that contain distinguished function symbols of Σ.

Definition 15 We denote by $\mathcal{C}$ a subset of Σ containing no constant symbol. A clause C is *strongly controlled* if it is controlled and contains no occurrence of symbols in $\mathcal{C}$.

We start by an informal description of our approach. The basic idea is that a non-controlled clause may be reduced to a controlled one by applying the superposition rule from a particular class of ground clauses, called $\mathcal{C}$-equations:

Definition 16 A $\mathcal{C}$-*equation* is a clause of the form $f(a_1,\dots,a_n) \simeq b \vee D$, where $a_1,\dots,a_n, b \in \Sigma_0, f \in \mathcal{C}$ and D is flat and ground.

Example 8 Consider the following axiom in the theory of arrays:

$$\forall x,z,v,w.\ z \simeq w \vee \mathtt{select}(\mathtt{store}(x,z,v),w) \simeq \mathtt{select}(x,w).$$

We have seen in the introduction of this section that this clause is not controlled. Assume that $\mathtt{store}$ is in $\mathcal{C}$. Because the theory of arrays is saturated, if the only clauses containing the function symbol $\mathtt{store}$ and not occurring in the axioms are $\mathcal{C}$-equations (i.e. are of the form $\mathtt{store}(a,i,e) \simeq b \vee D$ where D is flat and ground), then the only non-redundant inference that can be applied on the previous axiom is superposition from a $\mathcal{C}$-equation into $\mathtt{store}(x,z,v)$. The obtained clauses are of the form $\forall w.\ i \simeq w \vee \mathtt{select}(b,w) \simeq \mathtt{select}(a,w)$, and these clauses are controlled.

More generally, let S be a set of clauses. We can decompose the set of clauses that are deducible from S into three disjoint (possibly infinite) clause sets: a set of clauses S_{sc} that are strongly controlled (i.e. that are controlled and do not contain any symbol in $\mathcal{C}$), a set of $\mathcal{C}$-equations $S_{\mathcal{C}} \subseteq S$ and the remaining clauses S'. Initially, $S \subseteq S_{\mathrm{sc}} \cup S_{\mathcal{C}} \cup S'$.

The inferences within $S_{\mathrm{sc}} \cup S_{\mathcal{C}}$ are not problematic because they are simple by Theorem 5. Moreover, one can show that the obtained clauses are still in $S_{\mathrm{sc}} \cup S_{\mathcal{C}}$. This property is proven in Section 5.2.

The remaining inferences involve clauses in S': they are either inferences within S' or inferences between S' and $S_{\mathrm{sc}} \cup S_{\mathcal{C}}$. The inferences between S_{sc} and S' are hard to monitor because S_{sc} may be infinite. Thus we define syntactic conditions that ensure that no inference is possible between S_{sc} and S'. Informally, this is done by imposing that every eligible term in S' contains an occurrence of a function symbol in $\mathcal{C}$, at a

position where it cannot be unified with a variable. This obviously prevents unification with a term in $S_{\rm sc}$ because by definition $S_{\rm sc}$ contains no symbol in $\mathcal{C}$. The terms satisfying these restrictions are called *$\mathcal{C}$-restricted*. The formal definitions and results showing that no inference is possible between a clause with $\mathcal{C}$-restricted terms and one in $S_{\rm sc}$ are presented in Section 5.3.

Then we explicitly compute the set S' in order to check that (i) all eligible terms in S' are $\mathcal{C}$-restricted, (ii) no non-redundant inference is possible within S' (this is guaranteed, e.g., when S' is saturated). This is done as follows. We start from a clause set containing all the clauses in S that occur neither in $S_{\rm sc}$ nor in $S_{\mathcal{C}}$, and we check that (i) and (ii) hold. If this is the case then the only remaining inferences are those between clauses in $S_{\mathcal{C}}$ and clauses in S'. It turns out that those inferences always terminate since they replace a complex term by a constant symbol (by superposition from $\mathcal{C}$-equations). Thus it is possible to compute the entire set S' explicitly and check that (i) and (ii) still hold. Note that the clauses generated during the superposition process are not always added to S': they may actually occur in $S_{\rm sc}$ if they are strongly controlled. We do not need to test Conditions (i) and (ii) on such clauses, thus they are simply ignored. In other words, we apply superposition from $S_{\mathcal{C}}$ into S' until S' is *partially* saturated, in the sense that all the deducible clauses are in $S_{\rm sc}$. The corresponding algorithm is provided in Section 5.4. It is actually slightly more complicated than the one just sketched because we also take into account the fact that some inferences within S' may produce clauses that are actually redundant w.r.t. $S_{\mathcal{C}}$ (but not w.r.t. S'). Moreover we do not need to compute the clauses in S' themselves but only the part of these clauses that is not flat and ground. Thus, the only inferences are either those within $S_{\rm sc} \cup S_{\mathcal{C}}$ (the generated clause are necessarily in $S_{\rm sc} \cup S_{\mathcal{C}}$) or applications of the superposition/paramodulation rule into elements in S' from elements in $S_{\mathcal{C}}$ (the generated clauses are either in $S_{\rm sc}$ or in S'). All these inferences are simple.

Although finite, S' can be extremely large. Furthermore, it depends on the set of constant symbols, which is not acceptable in practice. However, an explicit computation of S' can be avoided, and this issue is discussed in Section 6.

5.2 Monitoring inferences on controlled clauses and $\mathcal{C}$-equations

The results of Section 4.2 on controlled clauses can be transposed to $\mathcal{C}$-equations and strongly controlled clauses. In this section we prove that the clauses that are deducible from strongly controlled clauses and $\mathcal{C}$-equations are necessarily strongly controlled, except if they are obtained from a $\mathcal{C}$-equation by ground flat superposition.

Lemma 8 *Let C and D be clauses, and σ denote a substitution:*

1. *If C is flat and ground then C is strongly controlled.*
2. *If C and D are strongly controlled, then so is $C \vee D$.*
3. *If σ is flat and C is a $\mathcal{C}$-equation (resp. a strongly controlled clause), then so is $C\sigma$.*

Proof Item 1 results from the fact that $\mathcal{C}$ only contains function symbols, and Item 2 is an immediate consequence of Lemma 5 (2). For Item 3, if C is a $\mathcal{C}$-equation then the result is obvious, since C is ground. Otherwise, the result is a direct consequence of Lemma 5 (3), since $\mathcal{C}$ only contains function symbols, and σ is a flat substitution.

Lemma 9 *Let C be a clause and let t, s be two non-variable terms such that:*

1. *s is I_0-flat, $\mathrm{I}_{\rm nv}$-closed, and contains no symbol in $\mathcal{C}$;*
2. *$I_{\mathcal{V}}(s) = I_{\mathcal{V}}(t)$;*

3. if $t \in \Sigma_0$, then $s \in \Sigma_0$.

If $C[t]_p$ is either a $\mathcal{C}$-equation or a strongly controlled clause, then so is $C[s]_p$.

Proof First assume that $C[t]_p$ is a $\mathcal{C}$-equation, and is therefore of the form $f(a_1, \ldots, a_n) \simeq b \vee C'$, where $a_1, \ldots, a_n, b \in \Sigma_0$, and C' is flat and ground. Necessarily, t is either a constant or the term $f(a_1, \ldots, a_n)$. If t is a constant, then $s \in \Sigma_0$ by hypothesis and $C[s]_p$ is also $\mathcal{C}$-equation. Otherwise, $t = f(a_1, \ldots, a_n)$, thus $C[s]_p = s \simeq b \vee C'$, and by Item 1, this clause is strongly controlled.

Now assume that C is a strongly controlled clause. Then by Lemma 6, $C[s]_p$ is controlled, and since s contains no symbol in $\mathcal{C}$ by hypothesis, $C[s]_p$ is strongly controlled.

Lemma 10 *Let D_1, D_2 denote (not necessarily distinct) clauses that are either $\mathcal{C}$-equations or strongly controlled clauses. If $\{D_1, D_2\} \rightarrow^{\sigma}_{\text{sel},<} C$, then C is either a $\mathcal{C}$-equation or a strongly controlled clause. Furthermore, if neither D_1 nor D_2 is flat and ground, then C is strongly controlled.*

Proof Assume that the clause C is generated by a unary rule from a $\mathcal{C}$-equation $D_1 = D_1' \vee f(a_1, \ldots, a_n) \simeq b$. If C is obtained by Reflection then, since D_1 is ground, by definition of the Reflection rule, C is of the from $D_1'' \vee f(a_1, \ldots, a_n) \simeq b$ where $D_1' = D_1'' \vee c \not\simeq c$, for some $c \in \Sigma_0$. Thus C is a $\mathcal{C}$-equation. If C is obtained by Eq. Factorisation then the literal on which the rule is applied must be selected. By Definition 11, this implies that this literal is maximal (since it must be positive by definition of the rule). Since we assume that $f(a_1, \ldots, a_n)$ is greater than any constant c (by the goodness property) this implies that this literal must be $f(a_1, \ldots, a_n) \simeq b$, and that the term t in the definition of the rule is $f(a_1, \ldots, a_n)$. But this is impossible since there is no other term in D_1 that is unifiable with $f(a_1, \ldots, a_n)$.

By Lemma 7, if neither D_1 nor D_2 is a $\mathcal{C}$-equation, then C is strongly controlled. Now assume that C is generated by the superposition or paramodulation rule, where D_1 or D_2, possibly both, is a $\mathcal{C}$-equation. This means that D_1 is of the form $L[t]_p \vee D_1'$, D_2 of the form $u \simeq v \vee D_2'$, and C is of the form $(L[v]_p \vee D_1' \vee D_2')\sigma$, where $\sigma = mgu(t, u)$.

If D_2 is a $\mathcal{C}$-equation, then necessarily, u is of the form $f(a_1, \ldots, a_n)$ where $f \in \mathcal{C}$, v is a constant, and D_2' is flat and ground. By hypothesis, D_1 is either a $\mathcal{C}$-equation or contains no symbols in $\mathcal{C}$. Thus, since u and t are unifiable, it must be a $\mathcal{C}$-equation of the form $u \simeq s \vee D_1'$, where $s \in \Sigma_0$, and D_1' is flat and ground. Therefore, in this case, $C = s \simeq v \vee D_1' \vee D_2'$ is flat and ground, hence strongly controlled.

If D_1 is a $\mathcal{C}$-equation and D_2 is not, then L must be the literal of the form $f(c_1 \ldots, c_n) \simeq d$, and t must be a subterm of $f(c_1, \ldots, c_n)$. We cannot have $t = f(c_1, \ldots, c_n)$ since otherwise u would have to contain a symbol in $\mathcal{C}$, thus t is a constant. Since D_2 is strongly controlled, it is also variable-preserving, and by Proposition 6, it is not variable-eligible. Thus, u is necessarily a constant, and D_2 must be flat by the conditions in Definition 11 for the selection function. Since D_2 is non-variable-eligible, it must be ground. In this case, C is also a $\mathcal{C}$-equation.

5.3 $\mathcal{C}$-restricted terms

The goal of $\mathcal{C}$-restricted terms is to control the inferences that can be performed on a given non-controlled clause C, in order to ensure that the following conditions hold: (i) Superposition from a $\mathcal{C}$-equation is possible, in order to transform C into a controlled clause; and (ii) No inference is possible between C and a controlled clause D that is not a $\mathcal{C}$-equation.

In order to guarantee that (i) holds, it suffices to assume that every eligible term t in C contains a subterm of the form $f(t_1, \ldots, t_n)$ where $f \in \mathcal{C}$, and $t_1, \ldots, t_n \in T_0$. For (ii), we exploit the fact that the term s in D that t is unified with is necessarily $\mathrm{I_{nv}}$-closed since D is controlled, thus there are some positions in s along which variables cannot occur (see Definition 3). We shall assume that the term $f(t_1, \ldots, t_n)$ occurs in t at such a position. By making sure D does not contain the symbol f, we shall prevent t and s from being unifiable (since t contains an occurrence of f at a position in which no variable occurs in s). This will guarantee that no inference is possible between C and D.

In the following definition, recall that $\mathrm{I_{nv}}(f)$ denotes the set of indices of f that must not be variables.

Definition 17 A term t is *$\mathcal{C}$-restricted* if and only if it is of the form $f(t_1, \ldots, t_n)$ where one of the following conditions holds:

1. $f \in \mathcal{C}$, and $f(t_1, \ldots, t_n)$ is of depth 1 and linear, i.e. contains at most one occurrence of each variable.
2. There exists at least an $i \in \mathrm{I_{nv}}(f)$ such that t_i is $\mathcal{C}$-restricted, and for all $j \in [1..n]$, if $j \in \mathrm{I_{nv}}(f)$ then either $t_j \in \Sigma_0$ or t_j is $\mathcal{C}$-restricted.

The fact that i is in $\mathrm{I_{nv}}(f)$ ensures that no inference is possible with a controlled term except with a $\mathcal{C}$-equation (see Lemma 11). The second subcondition in Point 2 ensures the term will become $\mathrm{I_{nv}}$-closed after all symbols in $\mathcal{C}$ have been eliminated by superposition. The fact that $f(t_1, \ldots, t_n)$ is linear is not really restrictive; furthermore it is useful for technical reasons (see the proof of Theorem 6): informally it ensures that superposition into $f(t_1, \ldots, t_n)$ can be restricted to the equations already occurring in the initial clause set: the replacement of constant symbols by ground flat superposition can be postponed to *after* the application of the superposition rule into $f(t_1, \ldots, t_n)$. This is useful to ensure that the derivation satisfies Condition 3 of Definition 9.

Example 9 Assume that $\mathcal{C} = \{f, g\}$ and that $\mathrm{I_{nv}}(h) = \{1\}$. Then:

- $f(x)$, $h(f(x), a)$, $h(f(x), g(x))$ and $h(h(g(x), y), x)$ are $\mathcal{C}$-restricted.
- $h(x, a)$ and $h(h(x, y), f(x))$ are not.

Proposition 7 *Let t be an $\mathrm{I_0}$-flat and $\mathcal{C}$-restricted term, and let $s = f(s_1, \ldots, s_n)$ be a subterm of t. If $s_i \in \mathcal{V}$ for some $i \in \mathrm{I_{nv}}(f)$, then $f \in \mathcal{C}$ and $s_1, \ldots, s_n \in T_0$.*

Proof The proof is by induction on the size of t. If $t = s$ then since we have $i \in \mathrm{I_{nv}}(f)$ and $s_i \in \mathcal{V}$, the second item in Definition 17 cannot hold. Thus we have $f \in \mathcal{C}$ and t is of depth 1 which completes the proof.

Otherwise, $t = g(t_1, \ldots, t_m)$ where s is a subterm of t_j, for some $j \in [1..m]$. Since s is a subterm of t_j, the latter cannot be in T_0, and since t is $\mathrm{I_0}$-flat, $j \notin \mathrm{I_0}(g)$. Since $\mathrm{I_0}(g) \cup \mathrm{I_{nv}}(g) = [1..m]$, we deduce that $j \in \mathrm{I_{nv}}(g)$. By hypothesis t is $\mathcal{C}$-restricted, hence t_j must be either a constant symbol or a $\mathcal{C}$-restricted term. Since s is a subterm of t_j, the latter cannot be a constant symbol, and is therefore a $\mathcal{C}$-restricted term. Since t is $\mathrm{I_0}$-flat, so is t_j, thus we can apply the induction hypothesis to obtain the result.

The following lemma allows to control the inferences that can be performed on $\mathcal{C}$-restricted terms.

Lemma 11 *Let $t \notin \mathcal{V}$ be an $\mathrm{I_{nv}}$-closed term such that for every position p in t distinct from ϵ, the head symbol of $t_{|p}$ is not in $\mathcal{C}$. Let s be a $\mathcal{C}$-restricted term of depth at least 2. Then t and s are not unifiable.*

Proof The proof is by induction on the size of s. By definition s must be of the form $f(s_1, \ldots, s_n)$. Since t is not a variable, we may assume that t is of the form $f(t_1, \ldots, t_n)$, otherwise t and s are obviously not unifiable. Since the depth of s is strictly greater than 1, the subterms $s_1, \ldots, s_n$ cannot all be flat. Thus by the second item of Definition 17, there exists an $i \in \mathrm{I_{nv}}(f)$ such that s_i is $\mathcal{C}$-restricted; necessarily, s_i is of depth at least 1. Assume that s_i is of depth 1. In this case, it must satisfy the conditions of the first item of Definition 17, thus its head symbol must be in $\mathcal{C}$. Since $i \in \mathrm{I_{nv}}(f)$ and t is $\mathrm{I_{nv}}$-closed, t_i cannot be a variable; furthermore, by hypothesis, the head symbol of t_i does not belong to $\mathcal{C}$, hence t_i and s_i are not unifiable. If s_i is of depth strictly greater than 1, then since t_i is $\mathrm{I_{nv}}$-closed, we may apply the induction hypothesis and conclude that s_i and t_i cannot be unifiable. Hence the result.

5.4 Monitoring superposition into non-controlled clauses

As was previously mentioned, we need to ensure that two non-controlled clauses cannot interfere with each other using one of the binary inference rules of the superposition calculus. We have to check that this property holds not only for the clauses occurring in S, but also for all clauses that can be derived by superposition. This is done by considering all the clauses that can be derived from S by superposition from $\mathcal{C}$-equations and from ground flat clauses. Such a set will be denoted by $[S]$. This set is finite (although very large), thus it is easy, at least in theory, to check whether it is saturated.

The equations used during the superposition process will be collected and explicitly added as "constraints" to the clause. This provides useful information and permits to discard some possible inferences from non-controlled clauses, if they are redundant with the $\mathcal{C}$-equations that were used to derive them. The following example illustrates this point.

Example 10 Consider the theory of lists, axiomatized by

$$S = \{\mathtt{cons}(\mathtt{car}(x), \mathtt{cdr}(x)) \simeq x,\ \mathtt{car}(\mathtt{cons}(x, y)) \simeq x,\ \mathtt{cdr}(\mathtt{cons}(x, y)) = y\},$$

and let $\mathcal{C} = \{\mathtt{cons}, \mathtt{car}, \mathtt{cdr}\}$. It is easy to verify that S is not controlled. In order to monitor the derivations starting from S, we need to consider the way the clauses it contains interact with $\mathcal{C}$-equations. Using the $\mathcal{C}$-equation $\mathtt{car}(a) \simeq b$ we can generate $\mathtt{cons}(b, \mathtt{cdr}(a)) \simeq a$. A problem arises: the obtained clause is still not controlled, and it interferes with the clause $\mathtt{car}(\mathtt{cons}(x, y)) = x$, which contradicts the property we require (no inference within non-controlled clauses). However, the clause generated by this inference is $\mathtt{car}(a) \simeq b$, and it is redundant with (actually identical in this case) the clause used to derive $\mathtt{cons}(b, \mathtt{cdr}(a)) \simeq a$.

In order to formalize this idea, we introduce the notion of e-clauses.

Definition 18 An *e-clause* is a pair $[C \mid \phi]$ where C is a clause and ϕ is a set (or conjunction) of equations. The *clausal part* of the e-clause is the clause C, and its *constraint part* is the set of equations ϕ.

The conjunction of equations ϕ in an e-clause $[C \mid \phi]$ intuitively denotes a set of literals that have been paramodulated into a clause that is not necessarily controlled,

to generate the clause C. The set ϕ will be used to harness the inferences that admit C as a premise. Note that e-clauses are similar to constrained clauses (see, e.g., [26]) but the equations in ϕ do not constrain the variables appearing in C.

The following relation computes new e-clauses by superposition from $\mathcal{C}$-equations and flat ground clauses, and simultaneously adds the corresponding equations to the set of constraints.

Definition 19 We denote by $\rightsquigarrow$ the smallest reflexive and transitive relation on e-clauses satisfying the following condition: For all e-clauses $[C \mid \phi]$ and for all terms $f(t_1, \ldots, t_n)$ occurring at a position p in C, if $f \in \mathcal{C} \cup \Sigma_0$ and $t_1, \ldots, t_n \in T_0$ then

$$[C \mid \phi] \rightsquigarrow [C[c]_p \mid \phi \cup \{t \simeq c\}]\theta,$$

where $c \in \Sigma_0$ and θ is a substitution mapping all variables in t to constant symbols.

Example 11 Consider the following axiom from the theory of arrays:

$$C : z \simeq w \vee \mathtt{select}(\mathtt{store}(x, z, v), w) \simeq \mathtt{select}(x, w).$$

Assume that $\mathtt{store} \in \mathcal{C}$. Then we have

$$[C \mid \emptyset] \rightsquigarrow [i \simeq w \vee \mathtt{select}(b, w) \simeq \mathtt{select}(a, w) \mid \mathtt{store}(a, i, e) \simeq b],$$

where a, b, i, e denotes distinct constant symbols.

The relation $\rightsquigarrow$ permits to generate e-clauses containing fewer symbols in $\mathcal{C}$ and to rewrite constant symbols. This relation resembles the flattening operation described previously, but is not applied to all the function symbols in the signature, and it also serves as a renaming operation for constants. Among the clauses that are generated by the relation, it will not be necessary to constrain those that are controlled. The whole set of e-clauses that can be deduced in this way, omitting controlled clauses, is denoted by $[S]$.

Definition 20 If S is a set of clauses, we denote by $[S]$ the smallest set of e-clauses such that the following conditions hold:

- If $C \in S$ and C is neither strongly controlled nor a $\mathcal{C}$-equation then $[C \mid \emptyset] \in [S]$.
- If $[C \mid \phi]$ is in $[S]$, $[C \mid \phi] \rightsquigarrow [D \mid \psi]$ and D is not strongly controlled, then $[D \mid \psi] \in [S]$.

Example 12 Let $S = \{\mathtt{cons}(\mathtt{car}(x), \mathtt{cdr}(x)) \simeq x\}$. Assume that $\mathtt{car}, \mathtt{cdr}, \mathtt{cons} \in \mathcal{C}$. The reader can check that $[S]$ consists of the following e-clauses:

1	$[\mathtt{cons}(\mathtt{car}(x), \mathtt{cdr}(x)) \simeq x \mid \emptyset]$	% the clause in S
2	$[\mathtt{cons}(b, \mathtt{cdr}(a)) \simeq a \mid \mathtt{car}(a) \simeq b]$	% from E-Clause 1
3	$[\mathtt{cons}(\mathtt{car}(a), b) \simeq a \mid \mathtt{cdr}(a) \simeq b]$	% from E-Clause 1
4	$[\mathtt{cons}(b, c) \simeq a \mid \mathtt{car}(a) \simeq b, \mathtt{cdr}(a) \simeq c]$	% from E-Clause 2
5	$[\mathtt{cons}(a, \mathtt{cdr}(a)) \simeq a \mid \mathtt{car}(a) \simeq b, b \simeq a]$	% from E-Clause 2
6	$[\mathtt{cons}(\mathtt{car}(a), a) \simeq a \mid \mathtt{cdr}(a) \simeq a, b \simeq a]$	% from E-Clause 3
7	$[\mathtt{cons}(b, b) \simeq a \mid \mathtt{car}(a) \simeq b, \mathtt{cdr}(a) \simeq c, c \simeq b]$	% from E-Clause 4
8	$[\mathtt{cons}(b, b) \simeq b \mid \mathtt{car}(a) \simeq b, \mathtt{cdr}(a) \simeq c, c \simeq b, a \simeq b]$	% from E-Clause 7

The remaining e-clauses are equivalent to the ones above, modulo a renaming of constant symbols. For instance, one could generate $[\mathtt{cons}(c, b) \simeq a \mid \mathtt{cdr}(a) \simeq b, \mathtt{car}(a) \simeq c]$

using E-Clause 3 and $\mathtt{car}(a) \simeq c$, but this e-clause is equivalent to E-Clause 4, up to a renaming of the constant symbols.

Note that the e-clause $[d \simeq a \mid \mathtt{car}(a) \simeq b, \mathtt{cdr}(a) \simeq c, \mathtt{cons}(b,c) \simeq d]$, that can be generated from E-Clause 4 (replacing $\mathtt{cons}(b,c)$ by d) does not occur in $[S]$ because it is strongly controlled.

We provide a link between the relations $\equiv_C^S$ of Definition 2 and $\rightsquigarrow$.

Proposition 8 *Let $[C \mid \phi]$ be an e-clause. If $C \equiv_D^S C'$ then $[C \mid \phi] \rightsquigarrow [C' \mid \phi \vee \psi]$, where $\forall e \in \psi$, e is flat and ground, and $e \vee D$ is redundant w.r.t. S.*

Proof This follows immediately from Definitions 2 and 19. Indeed, since $C \equiv_D^S C'$, this means several terms occurring in C are replaced to yield C', and if a constant t is replaced by s, then there is a clause $D' \subseteq D$ such that $t \simeq s \vee D'$ occurs in S. The equation $t \simeq s$ occurs in ψ, and obviously, $t \simeq s \vee D$ is subsumed by $t \simeq s \vee D'$.

5.5 Definition of the class of $\mathcal{C}$-controllable clauses

We define the class of $\mathcal{C}$-controllable sets of clauses, which, as we shall show, are simply provable.

Definition 21 A set of clauses S is *$\mathcal{C}$-controllable* if:

1. S is I_0-flat.
2. If $[C \mid \phi] \in [S]$ then C is not variable-eligible.
3. If $[C \mid \phi] \in [S]$ then every eligible term in C that is not a constant is $\mathcal{C}$-restricted.
4. If $[C \mid \phi]$ and $[D \mid \psi]$ are two (variable-disjoint renamings of) e-clauses in $[S]$ and if $\{C, D\} \rightarrow_{\mathrm{sel},<}^{\sigma} E$, then:
   - either E is redundant w.r.t. $\phi \cup \psi \cup S$,
   - or $\sigma = id$ and E is strongly controlled.

Example 13 Consider the following axioms from the theory of arrays.

$$
\begin{array}{ll}
(a_1) & \mathtt{select}(\mathtt{store}(x,z,v),z) \simeq v \\
(a_2) & z \simeq w \vee \mathtt{select}(\mathtt{store}(x,z,v),w) \simeq \mathtt{select}(x,w)
\end{array}
$$

Let $S = \{(a_1),(a_2)\} \cup S'$ where S' only contains flattened ground clauses. We have seen that $(a_2) \rightsquigarrow [i \simeq w \vee \mathtt{select}(b,w) \simeq \mathtt{select}(a,w) \mid \mathtt{store}(a,i,e) \simeq b]$, where a,b,i,e are constant symbols (see Example 11). Similarly, it is simple to verify that $(a_1) \rightsquigarrow [\mathtt{select}(b,i) \simeq e \mid \mathtt{store}(a,i,e) \simeq b]$. Assume that $\mathrm{I}_{vp}(\mathtt{select}) = \{2\}$. Then $I_{\mathcal{V}}(\mathtt{select}(a,w)) = I_{\mathcal{V}}(\mathtt{select}(b,w)) = \{w\}$, thus $i \simeq w \vee \mathtt{select}(b,w) \simeq \mathtt{select}(a,w)$ is variable-preserving. Furthermore, if $\mathrm{I}_0(\mathtt{select}) = \{2\}$ and $\mathrm{I}_{\mathrm{nv}}(\mathtt{select}) = \{1\}$ then it is simple to check that $i \simeq w \vee \mathtt{select}(b,w) \simeq \mathtt{select}(a,w)$ and $\mathtt{select}(b,i) \simeq e$ are controlled (hence strongly controlled since these clauses are not $\mathcal{C}$-equations). The clauses in S' are also controlled, since they are either flat and ground or of the form $f(a_1,\ldots,a_n) \simeq b \vee C$ where C is flat and ground. Therefore, we have $[S] = \{(a_1),(a_2)\}$. We now check that Conditions 1-4 of Definition 21 are satisfied.

1. All the terms in (a_1) and (a_2) are I_0-flat, since the second arguments of $\mathtt{select}$ are flat (and $\mathrm{I}_0(\mathtt{select}) = \{2\}$).
2. (a_1) and (a_2) are not variable-eligible, since the literal $z \simeq w$ cannot be selected (indeed, it is positive and non-maximal).

3. The only eligible terms are $\texttt{select}(\texttt{store}(x,z,v),z)$, $\texttt{select}(\texttt{store}(x,z,v),w)$ and $\texttt{store}(x,z,v)$, which are $\mathcal{C}$-restricted.
4. No non-redundant inference can be applied on $[S]$, thus Condition 4 holds.

Consequently, S is controllable.

Now, consider the following axiom, that defines symmetry of (bidimensional) arrays.
(a_3) $\texttt{symm}(x) \not\simeq \texttt{true} \vee \texttt{select}(\texttt{select}(x,z),w) \simeq \texttt{select}(\texttt{select}(x,w),z)$.

Let $S'' = \{(a_1),(a_2),(a_3)\} \cup S'$. The axiom (a_3) is I_0-flat and non-variable-eligible. However, (a_3) is not controlled, since for instance the first argument of $\texttt{select}(x,w)$ is a variable. Assume that $\texttt{symm}$ is in $\mathcal{C}$. If the selection function is arbitrary, then S'' is not controllable, since for instance the term $\texttt{select}(x,z)$ is obviously eligible and not $\mathcal{C}$-restricted, thus Condition 3 in Definition 21 is violated. However, if we assume that $\mathrm{sel}(a_3) = \{\texttt{symm}(t) \not\simeq \texttt{true}\}$ (which is possible since this literal is negative), then S'' becomes controllable. Indeed, $\texttt{symm}(x)$ is clearly $\mathcal{C}$-restricted. Furthermore, we have:

$$(a_3) \rightsquigarrow [b \not\simeq \texttt{true} \vee \texttt{select}(\texttt{select}(a,z),w) \simeq \texttt{select}(\texttt{select}(a,w),z) \mid \texttt{symm}(a) \simeq b].$$

Obviously, the clause $b \not\simeq \texttt{true} \vee \texttt{select}(\texttt{select}(a,z),w) \simeq \texttt{select}(\texttt{select}(a,w),z)$ is strongly controlled. Furthermore, the axiom (a_3) cannot interact with $(a_1),(a_2)$, because $\texttt{symm}(x) \not\simeq \texttt{true}$ is the only selected literal. Hence all the conditions of Definition 21 are fulfilled. This example shows the importance of the choice of both the set of symbols $\mathcal{C}$ *and* the selection function.

Condition 2 ensures that the clauses are not variable-eligible which is necessary to guarantee that a simple derivation exists. Condition 3 ensures that the only possible inferences are superpositions from $\mathcal{C}$-equations or flat, ground clauses, as explained in Section 5.3. Condition 4 ensures that no (non-trivial) inference is possible within the set. We make a useful exception (second item) for inferences yielding a clause that is controlled and that is not a $\mathcal{C}$-equation, provided it is ground. This is possible because ground inferences are always simple. This exception is useful for some theories because $[S]$ may contain $\mathcal{C}$-equations generated by superposition from non-controlled sets and interfering with each other. For instance, for the theory of lists, one can generate equations $\texttt{car}(a) \simeq b$ and $\texttt{car}(a) \simeq c$. These two equations interfere, yielding $b \simeq c$. However: (i) the result is strongly controlled, and (ii) the inference is ground. Relaxing the condition to allow non-ground inferences (e.g. flat inferences) would be incorrect as the following example shows.

Example 14 Let $S = \{p(x,f(x),y) \simeq \texttt{true}, p(u,u,f(v)) \not\simeq \texttt{true}, f(a) \simeq a\}$, where $f \in \mathcal{C}$. Then our instantiation scheme instantiates variables x,u,v with a, but not variable y. Indeed, the instantiation rule generates the following clauses:

1	$p(x,f(x),y) \simeq \texttt{true}$	(given)
2	$p(u,u,f(v)) \not\simeq \texttt{true}$	(given)
3	$f(a) \simeq a$	(given)
4	$p(u,u,f(a)) \not\simeq \texttt{true}$	(clause 2, using the term $f(a)$ from 3)
5	$p(a,f(a),y) \simeq \texttt{true}$	(clause 1, using the term $f(a)$ from 3)
6	$p(a,a,f(a)) \not\simeq \texttt{true}$	(clause 4, using the term $p(a,f(a),y)$ from 5)
7	$p(a,a,f(v)) \not\simeq \texttt{true}$	(clause 2, using the term $p(a,f(a),y)$ from 5)

The clause set obtained from this set by instantiating the remaining variables by λ is clearly satisfiable.

On the other hand, up to a renaming of the constant symbols, the set $[S]$ contains the clauses (we assume that $\mathrm{I}_{\mathrm{nv}}(p) = \{1, 2, 3\}$, thus these clauses are not controlled).

$$[S] = \{[p(c, c, y) \simeq \mathtt{true} \mid f(c) \simeq c], [p(c, d, y) \simeq \mathtt{true} \mid f(c) \simeq d]\} \\ \cup \{[p(u, u, d) \not\simeq \mathtt{true} \mid f(c) \simeq d]\},$$

and the only clause that can be generated from clauses in $[S]$ is $\square$, which is obviously strongly controlled, thus S would satisfy the conditions of Definition 21 if condition $\sigma = id$ were not included.

The conditions in Definition 21 are obviously decidable if S and Σ_0 are finite, since $[S]$ is finite in this case. The class of $\mathcal{C}$-controllable sets of clauses contains the class of controlled sets of clauses:

Proposition 9 *A set of clauses is controlled if and only if it is $\emptyset$-controllable.*

Proof Let S denote a set of clauses, and assume S is $\emptyset$-controllable. Then in particular, Condition 3 must hold for S. Thus, if there is an e-clause $[C \mid \phi]$ in $[S]$, then all the eligible terms in C must be $\mathcal{C}$-restricted. This is impossible since $\mathcal{C}$ is empty, thus, $[S]$ must be empty, and every clause in S must be controlled.

Conversely, if S is controlled, then $[S]$ is empty and Conditions 2, 3 and 4 trivially hold. By hypothesis, S is I_0-flat, and we have the result.

Another simple but important result is that a $\mathcal{C}$-controllable set of clauses remains so after a union with a ground set of clauses, modulo the flattening operation:

Proposition 10 *If S is $\mathcal{C}$-controllable and S' is a flattened set of ground clauses, then $S \cup S'$ is $\mathcal{C}$-controllable.*

Proof It suffices to remark that the clauses in S' are either flat, or of the form $f(a_1, \ldots, a_n) \simeq b$, where $a_1, \ldots, a_n, b$ are constants. Thus S' is trivially controlled, and the clauses in which symbols from $\mathcal{C}$ occur are $\mathcal{C}$-equations.

Thanks to this property, it is only necessary to test whether a theory is $\mathcal{C}$-controllable *once and for all*. Once this is done, the instantiation scheme is guaranteed to be correct, no matter what ground clauses are added to the theory, as long as they are flattened.

This result also permits us to consider applying the instantiation scheme in two ways: it can be applied *eagerly*, by instantiating the entire original problem, or it can be applied *lazily*, in a more DPLL(T)-like manner, on a conjunction of unit clauses that is a candidate model for the problem. It would be worthwhile to investigate which of the two manners is the more efficient.

5.6 Completeness for $\mathcal{C}$-controllable sets of clauses

In this section, we prove that every $\mathcal{C}$-controllable set of clauses is simply provable. In order to do so, we first introduce a class of clause sets that interact in a particular way with $\mathcal{C}$-controllable sets of clauses.

The set of clauses that can be generated by the superposition calculus starting from a $\mathcal{C}$-controllable set of clauses S may be infinite, and is not controlled in general. However, all these clauses satisfy the same property: either they are strongly controlled, or they are obtained from $\mathcal{C}$-equations in S or clauses occurring in $[S]$ by flat ground superposition. These conditions will be sufficient to guarantee the existence of simple

derivations starting from S. We define the clauses satisfying these conditions as S-*constricted*: intuitively, all important properties of an S-constricted set of clauses are consequences of the properties of $S \cup [S]$.

Definition 22 Let S be a set of clauses. A set of clauses S' is S-*constricted* if for all $C \in S'$ such that C is not strongly controlled, C is of the form $C_1 \vee C_2$ where C_2 is flat and ground, and there exists a clause D such that $D \equiv_{C_2}^{S'} C_1$ (see Definition 2) and:

1. either D is a $\mathcal{C}$-equation in S,
2. or $[S]$ contains an e-clause $[D \mid \phi]$ such that for every equation $(t \simeq s) \in \phi$, the clause $t \simeq s \vee C_2$ is redundant w.r.t. S'.

If Point 1 holds, this obviously implies that C is a $\mathcal{C}$-equation. Case 2 does not cover case 1 because $[D \mid \emptyset]$ does not occur in $[S]$ when $D \in S$ and D is controlled.

The following propositions state some useful properties of S-constricted clause sets.

Proposition 11 *If S_1 and S_2 are two sets of clauses that are S-constricted, then so is $S_1 \cup S_2$.*

Proposition 12 *Let S, S' be two sets of clauses. Suppose that S is $\mathcal{C}$-controllable and that S' is S-constricted. If C is a clause in S' that is not strongly controlled, then every eligible term t in C is either a constant or is $\mathcal{C}$-restricted.*

Proof By hypothesis, C is of the form $C_1 \vee C_2$, where C_2 is flat and ground, and there exists a clause D such that $D \equiv_{C_2}^{S'} C_1$. If D is a $\mathcal{C}$-equation then so is C, hence t must be of the form $f(c_1, \ldots, c_n)$ where $f \in \mathcal{C}$ and $c_1, \ldots, c_n \in \Sigma_0$, and the proof is obvious. Otherwise, by Point 2 of Definition 22, there exists a set of equations ϕ such that $[D \mid \phi] \in [S]$. By Proposition 8, $[D \mid \phi] \rightsquigarrow [C_1 \mid \phi \vee \psi]$ for some set of equations ψ. Since D is not strongly controlled, it must contain a symbol in $\mathcal{C}$, and so must C_1, which is not strongly controlled either. Thus by definition of $[S]$, the e-clause $[C_1 \mid \phi \vee \psi]$ is in $[S]$. If t occurs in C_2 then t is necessarily a constant. Otherwise, t is eligible in C_1 hence must be $\mathcal{C}$-restricted by Point 3 of Definition 21, since S is $\mathcal{C}$-controllable by hypothesis.

Proposition 13 *Let S, S' be two sets of clauses. Suppose that S is I_0-flat and that S' is S-constricted. Then every clause in S' is I_0-flat.*

Proof Let $C \in S'$. If C is strongly controlled then all its non-flat literals are controlled, hence I_0-flat by Definition 14. Otherwise, C is obtained from an I_0-flat clause by adding literals that are flat and ground, by instantiating some variables by constant symbols and by replacing some subterms by constant symbols. Obviously, the obtained clause is still I_0-flat.

Proposition 14 *Let S, S' be two sets of clauses. Suppose that S is $\mathcal{C}$-controllable and that S' is S-constricted. Then the clauses in S' are not variable-eligible.*

Proof Let $C \in S'$. We consider the conditions that may be satisfied by C:

- If C is controlled, then it is variable-preserving by Definition 14, and by Proposition 6, it is not variable-eligible.
- Otherwise C is of the form $C_1 \vee C_2$, where C_2 is flat ground and $\exists [D \mid \phi] \in [S]$ such that $D \equiv_{C_2}^{S'} C_1$. By Point (2) of Definition 21, D is not variable-eligible, thus neither are C_1 and C.

The next lemma shows that the application of an inference rule between two clauses satisfying Condition 2 of Definition 22 generates a clause satisfying strong properties.

Lemma 12 *Let S, S' be two sets of clauses such that S is $\mathcal{C}$-controllable and that S' is S-constricted. Let $D_1, D_2 \in S'$ be two clauses that satisfy Condition 2 of Definition 22, and are not flat and ground. If $\{D_1, D_2\} \rightarrow^{\sigma}_{\text{sel},<} C$, then one of the following conditions holds:*

- *C is redundant in $S \cup S'$;*
- *C is strongly controlled and $\sigma = id$;*

Proof By Point (2) of Definition 22 for $i = 1, 2$, D_i is of the form $D'_i \vee D''_i$ where D''_i is flat and ground, and there exists an e-clause $[E_i \mid \phi_i] \in [S]$ such that $E_i \equiv^{S'}_{D''_i} D'_i$, and $\forall e \in \phi_i$, $e \vee D''_i$ is redundant in S'. By definition of $[S]$, neither E_1 nor E_2 can be strongly controlled.

The flat and ground literals in D_1, D_2 cannot be selected, otherwise the clauses would be flat and ground. Therefore C must be of the form $C' \vee D''_1 \vee D''_2$, where $\{D'_1, D'_2\} \rightarrow^{\sigma}_{\text{sel},<} C'$. By Proposition 8, $[E_i \mid \phi_i] \rightsquigarrow [D'_i \mid \phi_i \cup \psi_i]$, where $\forall e \in \psi_i$, e is flat and ground and $e \vee D''_i$ is redundant in S'. Since E_1 and E_2 are not strongly controlled, neither are D'_1 and D'_2, thus for $i = 1, 2$, the e-clause $[D'_i \mid \phi_i \cup \psi_i]$ is in $[S]$.

Since S is $\mathcal{C}$-controllable, by Point (4) of Definition 21, one of the following conditions holds:

- C' is redundant in $\phi_1 \cup \phi_2 \cup \psi_1 \cup \psi_2 \cup S$. In this case, $C' \vee D''_1 \vee D''_2$ must be redundant in $\{e \vee D''_1 \vee D''_2 \mid e \in \phi_1 \cup \psi_1 \cup \phi_2 \cup \psi_2\} \cup S$. But for all $e \in \phi_1 \cup \psi_1 \cup \phi_2 \cup \psi_2$, $e \vee D''_1 \vee D''_2$ is redundant in S', hence C is redundant in $S \cup S'$ by Proposition 3.
- C' is strongly controlled and $\sigma = id$. Then C is also strongly controlled and satisfies the second condition of the lemma.

The following proposition states that S-constricted clause sets are stable by ground flat superposition.

Proposition 15 *Let S and S' be sets of clauses such that S is $\mathcal{C}$-controllable and S' is S-constricted. If $D_1, D_2 \in S'$, D_2 is flat and ground, and $\{D_1, D_2\} \rightarrow^{\sigma}_{\text{sel},<} C$, then σ is flat and $S' \cup \{C\}$ is S-constricted.*

Proof Assume that D_1 is flat. By Proposition 14 it cannot be variable-eligible, thus it must be ground by Proposition 2. Hence both D_1 and D_2 are flat and ground and so is C, thus C is strongly controlled by Lemma 8 (1), and $S' \cup \{C\}$ is S-constricted.

Now assume that D_1 is not flat. Then $D_1 \neq D_2$ hence C is deduced by superposition. Note that since D_1 is not variable-eligible, the selected literal in D_1 cannot be of the form $x \simeq t$ with $x \not< t$. This selected literal cannot be of the form $a \simeq b$ with a, b constant symbols either, since such a literal cannot be maximal in D_1 by definition of the ordering $<$.

If the superposition rule is applied from D_1 into D_2, then since D_2 is flat and ground, the selected term in D_1 must also be flat. But we have just seen that this case is impossible. Therefore, C is deduced by superposition from D_2 into D_1, and since there is no superposition into variables, C is of the form $D_1[b]_p \vee D'_2$, where $D_1|_p = a$ and $D_2 = a \simeq b \vee D'_2$. If D_1 is strongly controlled, then C is also strongly controlled. We now assume that D_1 is of the form $D'_1 \vee D''_1$ where D''_1 is flat ground and $D'_1 \equiv^{S'}_{D''_1} E$.

Necessarily, p must be a position in D'_1. Indeed, D_1 is not flat, hence the literals in D''_1 cannot be selected. Thus C is of the form $(D'_1[b]_q \vee D''_1 \vee D'_2)$. Furthermore,

$$D'_1[b]_q \equiv^{S'}_{D'_2} D_1[a]_q \equiv^{S'}_{D''_1} E,$$

thus $D'_1[b]_q \equiv^{S'}_{D''_1 \vee D'_2} E$. Finally, $D''_1 \vee D'_2$ is flat and ground, and if $e \vee D''_1$ is redundant w.r.t. S', then so is $e \vee D''_1 \vee D'_2$. This completes the proof.

We now prove that S-constricted clause sets are stable by superposition, and that all the inferences correspond to flat substitutions.

Lemma 13 *Let S and S' be sets of clauses such that S is $\mathcal{C}$-controllable and S' is S-constricted. Let S'' denote a set of pairwise variable-disjoint renamings of clauses in S'. If $S'' \rightarrow^{\sigma}_{\text{sel},<} C$ and C is not redundant in $S \cup S'$, then σ is flat and $S' \cup \{C\}$ is S-constricted.*

Proof We assume that no clause in S'' is flat and ground (otherwise the proof follows by Proposition 15). If all the clauses in S'' are controlled, then by Lemma 10, σ is flat and C is strongly controlled; hence $S' \cup \{C\}$ is S-constricted. If the clauses in S'' are not controlled, then they must satisfy Condition 1 of Definition 22, hence C is strongly controlled and σ is flat by Lemma 12 (because C is not redundant by hypothesis).

We now assume that one clause in S'' is controlled, and the other is not. Thus, S'' contains two clauses of the form $L[t]_p \vee D$ and $u \simeq v \vee D'$, and C is of the form $(L[v]_p \vee D \vee D')\sigma$, where σ is the mgu of t and u. Since no clause in S' is variable-eligible by Proposition 14, u is not a variable, and by definition of the calculus, neither is t. Furthermore, u cannot be a constant since, by definition of the ordering, that would imply $u \simeq v \vee D$ is flat and ground. Thus t and u are terms of the form $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$ respectively. We distinguish two cases, depending of which of the clauses in S'' is controlled.

The clause $L \vee D$ is controlled, but $u \simeq v \vee D'$ is not. We assume that either L is negative, or p is not a root position (otherwise the rôles of $L \vee D$ and $u \simeq v \vee D'$ can be swaped and the proof follows from the next point). By Definition 14, this implies that t is both I_0-flat and I_{nv}-closed, and that it cannot contain any symbol in $\mathcal{C}$. By Proposition 12, u must be $\mathcal{C}$-restricted. If u is of depth 1 then $f \in \mathcal{C}$, which is impossible since t contains no symbol in $\mathcal{C}$. Thus u is of depth at least 2 and by Lemma 11, t and u cannot be unifiable; we obtain a contradiction.

$u \simeq v \vee D'$ is controlled and $L \vee D$ is not. Since S' is S-constricted, by Definition 22, $L \vee D$ must be of the form $E_1 \vee E_2$, where E_2 is flat and ground, and there exists a clause E such that $E_1 \equiv^{S'}_{E_2} E$. Since $L \vee D$ is not a $\mathcal{C}$-equation, neither is E, thus, there exists an e-clause $[E \mid \phi] \in [S]$, and for all $e \in \phi$, $e \vee E_2$ is redundant w.r.t. S'. Obviously, L must occur in E_1, since L contains t which is not flat. Therefore, $L \vee D$ is of the form $L \vee D_1 \vee E_2$, where $E_1 = L \vee D_1$. Consequently, since $E_1 \equiv^{S'}_{E_2} E$, the latter is of the form $L' \vee D'_1$, where $L \equiv^{S'}_{E_2} L'$, and $D_1 \equiv^{S'}_{E_2} D'_1$.
Since $u \simeq v \vee D'$ is controlled, u contains no symbol in $\mathcal{C}$, except possibly at its root position if $u \simeq v \vee D'$ is a $\mathcal{C}$-equation. By Point (3) of Definition 21, every eligible term in E that is not a constant is $\mathcal{C}$-restricted; thus, in particular, t must be $\mathcal{C}$-restricted. Since t and u are unifiable, t must be of depth 1 by Lemma 11, and therefore, $f \in \mathcal{C}$. By Definition 14, this implies that $u \simeq v \vee D'$ is a $\mathcal{C}$-equation. Thus D' is flat and ground, and $v \in \Sigma_0$. Moreover, u is of the form $f(u_1, \ldots, u_n)$ where

$u_1, \ldots, u_n \in \Sigma_0$. Consequently, σ must be flat, and $x\sigma \in \Sigma_0$ for every variable x occurring in t.
Since $L' \vee D'_1 \equiv^{S'}_{E_2} L \vee D_1$, by Proposition 8 $[L' \vee D'_1 \mid \phi] \rightsquigarrow [L \vee D_1 \mid \phi \cup \psi]$, where $\forall e \in \psi, e \vee E_2$ is redundant in S'. Moreover, it is clear that $[L \vee D_1 \mid \phi \cup \psi] \rightsquigarrow [L[v]_p \vee D_1 \mid \phi \cup \psi \cup \{t \simeq v\}]\sigma$. By definition of $[S]$, since $[L' \vee D' \mid \phi] \in [S]$, either $[(L[v]_p \vee D_1)\sigma \mid \phi \cup \psi \cup \{t \simeq v\}]$ is strongly controlled, in which case C is also strongly controlled, or it is in $[S]$. Since $t\sigma = u$, the clause $(t\sigma \simeq v) \vee D'$ is in S'. Furthermore, for every $e \in \phi \cup \psi$, $e \vee E_2$ is redundant in S' (this follows immediately from the definition of ϕ and ψ), hence $C = L[v]_p \vee D_1 \vee E_2 \vee D'$ satisfies Condition 2 in Definition 22.

We are now in a position to prove the main result of this section:

Theorem 6 *If S is a $\mathcal{C}$-controllable set of clauses, then S is simply provable.*

Proof Let δ be a derivation of clause C starting from S, and let S_δ denote the set of clauses occurring in δ. We prove by induction on the length of δ that there exists a simple derivation δ' of C from S such that $S_{\delta'}$ is S-constricted.

If $C \in S$ then the proof is immediate: it suffices to take $\delta = \delta' = \varepsilon$. Assume now that C is deduced by applying an inference rule between two clauses D_1, D_2 (possibly with $D_1 = D_2$) using the substitution $\sigma = mgu(t, s)$, where t, s are terms occurring respectively in D_1 and D_2. Let δ_1, δ_2 be the derivations of D_1, D_2, and let $S' = S_{\delta_1} \cup S_{\delta_2}$. We may assume that C is not redundant w.r.t. $S \cup S'$, since otherwise the inference is useless.

By the induction hypothesis, we may assume that δ_1, δ_2 are simple, and that S_{δ_1} and S_{δ_2} are both S-constricted. This implies that S' is S-constricted by Proposition 11. By Lemma 13, σ is flat and $S' \cup \{C\}$ is S-constricted; thus $S_\delta = S' \cup \{C\}$ is also S-constricted. By Proposition 14, C is not variable-eligible, and by Proposition 13, C is I_0-flat. Hence Conditions 1 and 2 of Definition 9 hold for δ.

Assume that Condition 4 does not hold, i.e. that a superposition rule is applied from a clause of the form $D_2 = u \simeq v \vee D'_2 \in S'$, where v is a variable. In this case, $u \simeq v \vee D'_2$ cannot be variable-preserving, hence this clause is not controlled. But we have seen in the proof of Lemma 13 that superposition from a non-controlled clause is impossible.

There remains to prove that Condition 3 of Definition 9 is satisfied. This is actually not the case for δ in general. When this is not the case, we show how to construct another derivation δ' with the same root as δ, and such that Condition 3 holds for δ'.

Assume that $t|_p = f(t_1, \ldots, t_n)$, where $t_i \in \mathcal{V}$, for some $i \in \mathrm{I_{nv}}(f)$; the other case is symmetrical. This implies that t is not $\mathrm{I_{nv}}$-closed, i.e. that D_1 is not controlled. By Proposition 12, t is therefore $\mathcal{C}$-restricted. Furthermore, since S' is S-constricted, necessarily, D_1 must satisfy Point 2 of Definition 22. If D_2 satisfies Point 2 of Definition 22, then by Lemma 12, we must have $\sigma = id$, since C is not redundant in $S \cup S'$. Thus, Condition 3 of Definition 9 holds for δ which is simple. We now assume that D_2 does not satisfy Point 2 of Definition 22; this implies that D_2 is controlled.

By Proposition 7, since t is $\mathcal{C}$-restricted and I_0-flat, $t|_p$ must be of depth 1 and f must belong to $\mathcal{C}$. Since D_2 is controlled, s cannot be variable-eligible in D_2, and the symbol f must occur in s. Thus, since $f \in \mathcal{C}$ occurs in D_2, the latter must be a $\mathcal{C}$-equation, and be of the form $f(s_1, \ldots, s_n) \simeq c \vee E$, where $s = f(s_1, \ldots, s_n)$. Therefore, C must be of the form $(D_1[c]_q \vee E)\sigma$. Furthermore, since S' is S-constricted, D_2 satisfies Point (1) of Definition 22, so that S contains a clause $D'_2 = f(s'_1, \ldots, s'_n) \simeq c' \vee E'$

such that $D'_2 \equiv_E^{S'} D_2$. In particular, for all $j \in [1..n]$, $s_j \equiv_E^{S'} s'_j$, hence $s_j \simeq s'_j \vee E$ is redundant w.r.t. S'. For $j \in [1..n]$, if $t_j \in \Sigma_0$, then we must have $s_j = t_j$, since t, s are unifiable, and $s_1, \ldots, s_n \in \Sigma_0$. If $t_j \neq s'_j$, then t_i can be replaced by s'_i by an unordered superposition from the clause $s_j \simeq s'_j \vee E$. We obtain a term that is necessarily unifiable with $f(s'_1, \ldots, s'_n)$, since t is linear by Definition 17. Consequently, we can apply the superposition rule between D_1 and the resulting clause, to obtain a clause that subsumes $(D_1[c']_q \vee E \vee E')\sigma$. Using the superposition rule again on constant symbols, we can transform E' into E, since $E \equiv_E^{S'} E'$, and c' into c, since $c \equiv_E^{S'} c'$. Therefore, the derivation thus constructed generates C. In this derivation, t is unified with the term $f(s'_1, \ldots, s'_n)$ that occurs in S. Thus Condition 3 holds for this inference. The additional inference steps are ground, thus trivially satisfy the desired conditions. This proves that the obtained derivation δ' is simple, and $S_{\delta'}$ is S-controlled, since the clauses in $S_{\delta'}$ are obtained from clauses in S_δ by replacing constant symbols by other constant symbols.

6 Implementation and examples

6.1 Algorithms

Testing whether a set S is $\mathcal{C}$-controllable is not an obvious task, since it requires determining the subset of function symbols $\mathcal{C} \subseteq \Sigma$ and the sets of indices $\mathrm{I}_0(f), \mathrm{I}_{\mathrm{nv}}(f), \mathrm{I}_{vp}(f)$ for every function symbol $f \in \Sigma$, in order to compute the set $[S]$ and verify that the conditions of Definition 21 are satisfied. This task can of course be automated, and has been implemented at `http://membres-lig.imag.fr/peltier/fish.html`. A CAML-like pseudo-code description of the algorithm is provided in Figure 4. Our current program only implements the procedure Is_C-Controllable_aux, i.e. I_{vp}, I_0, I_{nv} and $\mathcal{C}$ are assumed to be given (as well as the selection function and the ordering). This information must be provided in the input file, together with the theory (see for instance `http://membres-liglab.imag.fr/peltier/smt`).

The algorithm of Figure 4 is a direct translation of the previous definitions, and can clearly be optimized. A few obvious optimizations can be applied to procedure Is_C-Controllable(S). For example, any function symbol f that never occurs in a non-ground term can be handled immediately: it suffices to set $\mathrm{I}_{vp}(f) = \mathrm{I}_0(f) = \emptyset$ and $\mathrm{I}_{\mathrm{nv}}(f) = [1..n]$, and there is no need to consider the subsets $\mathcal{C}$ that contain f. As it is defined, procedure Compute_E-Clauses(S) is hugely inefficient because of the nested **for** loops, which generate a very large set, especially if Σ contains many constant symbols. In practice, it is actually useless to generate S' explicitly. Instead, in our implementation, $[S]$ is computed in a symbolic way, replacing constant symbols by special variables, and the conditions are checked directly on this abstract representation of $[S]$ rather than on $[S]$ itself, taking into account the fact that the constants can be renamed arbitrarily. This affects both the unification algorithm (constant symbols are allowed to be substituted by other constant symbols) and the selection function (one has to check that a term is eligible modulo a renaming of constant symbols). A formal treatment of this optimization is out of the scope of this paper.

6.2 $\mathcal{C}$-Controllable Theories

Using our implementation, we were able to prove that our instantiation scheme can be safely applied to solve SMT problems in *all the theories defined in [3, 2]*. Below are listed some of the theories that can be handled by our scheme; they have all been checked automatically by our implementation. Appendix D contains the corresponding input file, which gives in particular the list of symbols in $\mathcal{C}$ and the value of I_0, I_{vp}

```
Is_C-Controllable(S) =
  for all sets I_vp(f), I_0(f), I_nv(f) associated to symbol f of arity n
    such that I_vp(f) ⊆ [1..n] and I_0(f) ∪ I_nv(f) = [1..n] do
  for all sets 𝒞 ⊆ Σ \ Σ_0 do
    if Is_C-Controllable_aux(S,I_vp,I_0,I_nv,𝒞) then return true
  return false

Is_C-Controllable_aux(S,I_vp,I_0,I_nv,𝒞) =
    // test of Condition 1
    if S is not I_0-flat then return false
      else let S' = Compute_E-Clauses(S) in
        for all [C | φ] in S' do
        // test of Conditions 2 and 3
          if C is variable eligible or
            C contains an eligible term that is not 𝒞-restricted then
            return false
        for all [C | φ], [D | ψ] in S' such that {C, D} →^σ_{sel,<} E do
        // test of Condition 4
          if (E is not redundant w.r.t. φ ∪ ψ ∪ S) and
            (σ ≠ id or E is not strongly controlled) then
            return false
  return true

Compute_E-Clauses(S) =
  let S' = {[C | ∅] | C ∈ S is not strongly controlled and not a 𝒞-equation}
  and T = S' in
    while T ≠ ∅ do
    let [C | φ] ∈ T in
      T := T \ {C}
      for all terms t of depth at most 1 occurring at position p in C do
        if the head symbol of t is in 𝒞 ∪ Σ_0 then
        for all flat ground substitutions θ of domain V(t),
        for all constants c ∈ Σ_0 do
          let [D | ψ] = [C[c]_p | φ ∪ {t ≃ c}] in
          if D is not strongly controlled then
            S' := S' ∪ {[D | ψ]}
            T := T ∪ {[D | ψ]}
    return S'
```

Fig. 4 Testing whether a set is $\mathcal{C}$-controllable

and I_{nv} for each function symbol (as well as the definition of the selection function). Due to the many constraints and restrictions, it is easy to check that the choice of the parameters is more or less unique (except on some minor aspects, e.g. the definition of I_{vp} is actually required only for symbols whose semantics are defined by axioms involving equations between variables, such as the function `select` of the theory of arrays). The intuition behind the choice of the indexes in I_0 and I_{nv} should be clear from the definitions and explanations in Section 2.3: $\mathrm{I}_0(f)$ is the set of indexes of the arguments of f that should be flat (variable or constant symbols), and $\mathrm{I}_{\mathrm{nv}}(f)$ is the set of indexes that should not be variables. $\mathrm{I}_{vp}(f)$ can be viewed as the set of indexes i such that t_i cannot be replaced in a term $f(t_1, \dots, t_n)$ by flat superpositions, without being instantiated; the variables in these terms are thus guaranteed never to become eligible in the clause generated by this superposition.

Intuitively, $\mathcal{C}$ may be viewed as a set of *constructors*: equations on terms with head symbol $f \in \mathcal{C}$ are restricted to ground terms already occurring in the initial clause set

(i.e. to constant symbols, since the set is flattened). Thus the axioms can only assert *properties* of these ground terms (for instance Axiom (a_3) states that an array t such that $\mathtt{symm}(t) \simeq \mathtt{true}$ is indeed symmetric) but cannot derive new equations on such symbols (except those involving already existing terms, e.g. $\mathtt{car}(a) \simeq b$ can be derived from $b \simeq \mathtt{cons}(a, c)$).

Natural Numbers

$$\begin{array}{ll} (n_1) & 0 \not\simeq \mathtt{succ}(x) \\ (n_2) & x \simeq y \vee \mathtt{succ}(x) \not\simeq \mathtt{succ}(y) \\ (n_3) & 0 < \mathtt{succ}(x) \\ (n_4) & x \not< y \vee \mathtt{succ}(x) < \mathtt{succ}(y) \end{array}$$

Integer Offsets

$$\begin{array}{lll} (i_1) & p(s(x)) \simeq x & \\ (i_2) & s(p(x)) \simeq x & \\ (i_3) & s^{n-i}(x) \not\simeq p^i(x) & n > 0 \end{array}$$

The theory of Integer Offsets Modulo k can also be handled.

Ordering

$$\begin{array}{ll} (o_1) & x \not\prec y \vee y \not\prec z \vee x \prec z \\ (o_2) & x \not\prec y \vee y \not\prec x \end{array}$$

Arrays

$$\begin{array}{ll} (a_1) & \mathtt{select}(\mathtt{store}(x, z, v), z) \simeq v \\ (a_2) & z \simeq w \vee \mathtt{select}(\mathtt{store}(x, z, v), w) \simeq \mathtt{select}(x, w) \\ (a_3) & \mathtt{symm}(x) \not\simeq \mathtt{true} \vee \mathtt{select}(\mathtt{select}(x, z), w) \simeq \mathtt{select}(\mathtt{select}(x, w), z) \\ (a_4) & \mathtt{injective}(x) \not\simeq \mathtt{true} \vee z \simeq w \vee \mathtt{select}(x, z) \not\simeq \mathtt{select}(x, w) \end{array}$$

The theory of records can be handled in a similar way. The literals $\mathtt{symm}(t) \not\simeq \mathtt{true}$ and $\mathtt{injective}(t) \not\simeq \mathtt{true}$ *must* be selected in the clauses above.

Encryption

$$\begin{array}{ll} (e_1) & \mathtt{dec}(\mathtt{enc}(x, y), y) = x \\ (e_2) & \mathtt{enc}(\mathtt{dec}(x, y), y) = x \end{array}$$

These axioms encode encryption and decryption with a symmetric key. The encryption operation takes a clear-text and a key and produces a cipher-text. The decryption operation inverses the encryption by extracting a clear-text from a cipher-text using the same key.

(Possibly Empty) Lists

$$\begin{array}{ll} (l_1) & \mathtt{car}(\mathtt{cons}(x, y)) \simeq x \\ (l_2) & \mathtt{cdr}(\mathtt{cons}(x, y)) \simeq y \\ (l_3) & x \simeq \mathtt{nil} \vee \mathtt{cons}(\mathtt{car}(x), \mathtt{cdr}(x)) \simeq x \\ (l_4) & \mathtt{cons}(x, y) \not\simeq \mathtt{nil} \end{array}$$

(Possibly Empty) Doubly Linked Lists

$$
\begin{array}{ll}
(ll_1) & x = \mathtt{nil} \vee \mathtt{next}(x) = \mathtt{nil} \vee \mathtt{prev}(\mathtt{next}(x)) = x \\
(ll_2) & x = \mathtt{nil} \vee \mathtt{prev}(x) = \mathtt{nil} \vee \mathtt{next}(\mathtt{prev}(x)) = x \\
(ll_3) & \mathtt{prev}(x) \simeq \mathtt{nil} \vee \mathtt{prev}(y) \simeq \mathtt{nil} \vee x \simeq y \vee x \simeq \mathtt{nil} \vee y \simeq \mathtt{nil} \\
 & \vee \mathtt{prev}(x) \not\simeq \mathtt{prev}(y) \\
(ll_4) & \mathtt{next}(x) \simeq \mathtt{nil} \vee \mathtt{next}(y) \simeq \mathtt{nil} \vee x \simeq y \vee x \simeq \mathtt{nil} \vee y \simeq \mathtt{nil} \\
 & \mathtt{next}(x) \not\simeq \mathtt{next}(y)
\end{array}
$$

ll_3 and ll_4 are logical consequence of ll_1, ll_2, but it is necessary to include these axioms explicitly in order to satisfy the conditions of Definition 21.

Other recursive data-structures can be handled in a similar way.

Others

The union of these theories is also $\mathcal{C}$-controllable. Furthermore, if a theory $\mathcal{T}$ is $\mathcal{C}$-controllable, and if S is a set of clauses containing only $\mathcal{C}$-equations and ground flat clauses, then $\mathcal{T} \cup S$ is also $\mathcal{C}$-controllable (see Proposition 10). This property is essential in practice, because one only has to check the controllability condition once and for all on $\mathcal{T}$.

One interesting point is that the superposition calculus does not necessarily terminate on $\mathcal{C}$-controllable clause sets. For instance, $g(x) \simeq f(g(y))$ is obviously $\mathcal{C}$-controllable (even controlled), but the superposition calculus deduces an infinite number of clauses of the form $g(x) \simeq f^n(g(y))$. This shows evidence of the interest of the instantiation method introduced in Section 3.

Sometimes non-controlled clauses may be reduced to controlled ones by simple equivalence-preserving transformations. For instance, assume that the signature is sorted and that an ordering $\prec$ on the sort symbols is given. Assume furthermore that there exists a sort s such that for every function $f : s_1 \times \ldots \times s_n \to s$ and for every $i \in [1..n]$: $s_i \prec s$. Then every quantified formula $\forall x : s.\phi$ can be replaced by the (equivalent) conjunction: $\bigwedge_{f:\mathbf{s} \to s} \forall \mathbf{x} : \mathbf{s}.\phi\{x \to f(\mathbf{x})\}$. f ranges over the set of functions of range s and $\mathbf{x}$ denotes a vector of distinct fresh variables of sort $\mathbf{s}$. This process necessarily terminates since a variable of sort s is replaced by variables of sorts strictly lower than s.

If the previous property holds for every sort symbol (i.e. the clause set is *stratified* in the sense of [1]), then the clause set can be transformed into an equivalent controlled one by repeating this transformation for every non-minimal sort. It is easy to check that the obtained formula is controlled (it suffices to take for every function symbol of arity n, $\mathrm{I}_{\mathrm{nv}}(f) = [1..n]$ if the range of f is not minimal and $\mathrm{I}_0(f) = [1..n]$ otherwise).

Another interesting case is when S contains terms of depth at most 1. S is necessarily I_0-flat and I_{nv}-closed (it suffices to take $\mathrm{I}_0(f) = [1..n]$ for every symbol f of arity n). Then S is controlled if and only if it is variable-preserving. Thus the class of variable-preserving clauses of depth 1 is decidable, although the superposition calculus does not terminate on this class and the corresponding Herbrand universe is infinite in the general case.

6.3 An Example of Application: the Theory of Arrays

It is worth investigating how our approach applies to well-known theories, such as the theory of arrays, for which efficient instantiation schemes are already known [11]. Let ϕ be a ground formula built on a signature containing only constant symbols

and the functions $\texttt{select}$ and $\texttt{store}$, as defined by Axioms (a_1) and (a_2). By flattening and (structural) transformation into clausal normal form, ϕ can be reduced to an equisatisfiable set of ground clauses $S_1 \cup S_2$ where S_2 is flat and S_1 only contains equations of the form $\texttt{select}(a, i) \simeq e$ and $\texttt{store}(a, i, e) \simeq b$ (a, b, i, e are constant symbols). As seen before, $\{(a_1), (a_2)\} \cup S_1 \cup S_2$ is $\mathcal{C}$-controllable. The instantiation scheme can be applied as follows. Obviously, the instantiation rule (I) applies on the clauses (a_1) and (a_2), using the terms $\texttt{store}(x, z, v)$ and $\texttt{store}(a, i, e)$ and generates instances $\texttt{select}(\texttt{store}(a, i, e), i) \simeq e$ and $i \simeq w \vee \texttt{select}(\texttt{store}(a, i, e), w) \simeq \texttt{select}(a, w)$, for every equation of the form $\texttt{store}(a, i, e) \simeq b$ occurring in S_2. This step is rather standard and similar to what is done in existing approaches (see for instance [11,7]). Then the rule may be applied to the term $\texttt{select}(a, w)$, to generate the following ground clause: $i \simeq i' \vee \texttt{select}(\texttt{store}(a, i, e), i') \simeq \texttt{select}(a, i')$, for each previously generated clause and for each term of the form $\texttt{select}(a', i') \simeq e'$ occurring in S_1. If the clauses are normalized during the instantiation step, replacing the terms $\texttt{store}(a, i, e)$ by b, then one could obtain instead: $\texttt{select}(b, i) \simeq e$ and $i \simeq i' \vee \texttt{select}(b, i') \simeq \texttt{select}(a, i')$.

Note that in contrast to other existing approaches, variable w is not instantiated by all possible indexes, but only by those really occurring in a $\texttt{select}$ term. This is (probably) not very significant if the instantiation scheme is applied eagerly (i.e. once and for all on the initial clause set), because in this case it is very likely that every constant of sort index will appear as the second argument of a $\texttt{select}$ (besides pathological cases). However, if the instantiation scheme is applied lazily then since the current branch does not contain in general all literals of head $\texttt{select}$, many potential indexes may be discarded.

Obviously, the instantiation rule can also be applied the other way round. First the term $\texttt{select}(\texttt{store}(x, z, v), z)$ in (a_1) can be pseudo-unified with $\texttt{select}(a', i')$, yielding a clause $\texttt{select}(\texttt{store}(x, i', v), i') \simeq v$, and then the rule can be applied to the term $\texttt{store}(x, i', v)$, which produces: $i \simeq i' \vee \texttt{select}(\texttt{store}(a, i', e), i') \simeq e$. But obviously the resulting clause is less general than $\texttt{select}(\texttt{store}(a, i, e), i) \simeq e$, thus it is redundant.

To summarize, the non-ground axioms (a_1) and (a_2) can be replaced by the ground set consisting of (a_1) and (a_2) instantiated by λ, along with $\texttt{select}(\texttt{store}(a, i, e), i) \simeq e$ and $i \simeq i' \vee \texttt{select}(\texttt{store}(a, i, e), i') \simeq \texttt{select}(a, i')$ (or, if clauses are normalized on the fly, by $\texttt{select}(b, i) \simeq e$ and $i \simeq i' \vee \texttt{select}(b, i') \simeq \texttt{select}(a, i')$), for every equations $\texttt{store}(a, i, e) \simeq b$ and $\texttt{select}(b', i') \simeq e'$ occurring in S_1.

6.4 Experimental Results

We illustrate our approach by applying it on a standard example (see for instance [2]), namely the commutativity of the store operation: the result of storing a set of elements at distinct positions in an array does not depend on the ordering on the elements.
Formally, if γ is a permutation of $[1..n]$, then $\{(a_1), (a_2), \bigwedge_{0 \leq j < k \leq n} i_j \not\simeq i_k\} \models t_n \simeq s_n$, where t_n and s_n are defined inductively as follows:

- $t_0 \stackrel{\text{def}}{=} s_0 \stackrel{\text{def}}{=} t$,
- $t_{k+1} \stackrel{\text{def}}{=} \texttt{store}(t_k, i_{k+1}, u_{k+1})$,
- $s_{k+1} \stackrel{\text{def}}{=} \texttt{store}(s_k, i_{\gamma(k+1)}, u_{\gamma(k+1)})$.

This is encoded by considering 4 lists of constant symbols: $i_1, \ldots, i_n$, $j_1, \ldots, j_n$, $u_1, \ldots, u_n$, $v_1, \ldots, v_n$. Intuitively j_k and v_k denote $i_{\gamma(k)}$ and $u_{\gamma(k)}$ respectively. One has to state that $(j_k, v_k)_{k \in [1..n]}$ is a permutation of $(i_k, v_k)_{k \in [1..n]}$ which is stated as follows: $\bigwedge_{l=1}^{n} \bigwedge_{k=i+1}^{n} (i_l \not\simeq i_k) \wedge \bigwedge_{l=1}^{n} \bigvee_{k=1}^{n} (i_l \simeq j_k) \wedge (u_l \simeq v_k)$.

Since there are exponentially many possible distinct permutations, the difficulty of the problem increases very quickly. For instance, for $n = 2$, we obtain the conjunction of the following formulae:

$$(a_1)$$
$$(a_2)$$
$$\mathtt{store}(\mathtt{store}(t, i_1, u_1), i_2, u_2) \simeq \mathtt{store}(\mathtt{store}(t, j_1, v_1), j_2, v_2)$$
$$i_1 \not\simeq i_2$$
$$[(i_1 \simeq j_1) \wedge (u_1 \simeq v_1)] \vee [(i_1 \stackrel{\text{def}}{=} j_2) \wedge (u_1 \simeq v_2)]$$
$$[(i_2 \simeq j_1) \wedge (u_2 \simeq v_1)] \vee [(i_2 \stackrel{\text{def}}{=} j_2) \wedge (u_2 \simeq v_2)]$$

We provide the results (in seconds) obtained with our instantiation scheme combined with the SMT-solver Yices [19] which is used to check the validity of the resulting set of ground instances (in the empty theory) and with Yices alone (Yices uses the techniques described in [11] for handling the theory of arrays). The instantiation time is included, but it is very short (a few ms), even on the larger instances.

n	(I)+Yices	Yices alone
5	0.078	0.063
7	0.391	0.344
8	1.234	1.094
9	7.781	6.828
10	38.375	98.563
11	1033.06	2018.5

As can be seen from the above table, our instantiation scheme (applied eagerly) is fully competitive with Yices's lazy instantiation technique. It is even more efficient on harder instances. This may be due to the fact that when n is large enough, the number of branches in the DPLL tree becomes so important that it is more efficient to perform the instantiation once and for all at the root level.

Of course, our generic approach is more relevant and useful when applied on theories (or combination of theories) for which no semantic instantiation scheme is available. Consider the following example. Obviously, a symmetric array stays so when the following operations are performed:

- Replacement of the elements on the diagonal by arbitrary values.
- Replacement of the symmetrical elements at positions $[i, j]$ and $[j, i]$ by the *same* value.

The formula below encodes these properties for a sequence of n operations. Starting from an array t, we successively replace the element at position $[e_i, e'_i]$ by v_i and the element at position $[e'_i, e_i]$ by v'_i, where for every $i \in [1..n]$, either $e_i = e'_i$ or $v_i = v'_i$. More precisely, for every $n \in \mathbb{N}$, we have

$$\{(a_1), (a_2), (a_3), \mathtt{symm}(t), E\} \models \mathtt{select}(\mathtt{select}(t_n, z), w) \simeq \mathtt{select}(\mathtt{select}(t_n, w), z),$$

where:

- $E \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} e_i \simeq e'_i \vee v_i \simeq v'_i$;
- t_n is defined as follows:
  - $t_0 \stackrel{\text{def}}{=} t$,
  - $t'_i \stackrel{\text{def}}{=} \mathtt{store}(t_i, e_{i+1}, \mathtt{store}(\mathtt{select}(t_i, e_{i+1}), e'_{i+1}, v_{i+1}))$.

– $t_{i+1} \stackrel{\text{def}}{=} \texttt{store}(t'_i, e'_i, \texttt{store}(\texttt{select}(t'_i, e'_i), e_i, v'_i))$,

Bidimensional arrays are encoded as arrays whose elements are 1-dimensional arrays. Of course, it would be possible (and actually much more natural and convenient) to extend the axioms $(a_1), (a_2)$ in order to handle pairs of indexes. We prefer to use bidimensional arrays to make the problem artificially more difficult. Solving the same problem with bidimensional arrays is actually a trivial task, since only one nonredundant instance of Axiom (a_3) may be generated (both our instantiation scheme and Yices's quantifier elimination heuristics detect this).

For instance, for $n = 2$, we obtain:

$$
\begin{array}{c}
(a_1) \\
(a_2) \\
(a_3) \\
\texttt{symm}(t) \\
e_1 \simeq e'_1 \vee v_1 \simeq v'_1 \\
e_2 \simeq e'_2 \vee v'_2 \simeq v'_2 \\
t'_0 = \texttt{store}(t, e'_1, \texttt{store}(\texttt{select}(t, e_1), e'_1, v_1)) \\
t_1 = \texttt{store}(t'_0, e_1, \texttt{store}(\texttt{select}(t, e'_1), e_1, v'_1)) \\
t'_1 = \texttt{store}(t_1, e'_2, \texttt{store}(\texttt{select}(t, e_2), e'_2, v_2)) \\
t_2 = \texttt{store}(t'_1, e_2, \texttt{store}(\texttt{select}(t, e'_2), e_2, v'_2)) \\
\texttt{select}(\texttt{select}(t, i), j) \not\simeq \texttt{select}(\texttt{select}(t, j), i)
\end{array}
$$

The results are summarized in the following table. They show that our instantiation scheme is less efficient than the one used by Yices on this example (however completeness is not guaranteed for the latter).

n	(I)+Yices	Yices alone
3	0.093	0.078
4	0.14	0.312
5	3.094	1.672
6	19.844	14.062
7	82.703	40.703
8	255.718	88.094

We now provide a simple example that is outside the scope of Yices's quantifier elimination heuristics. From an n-dimensional array t, one can construct n arrays of dimension $n-1$: $t'_1, \ldots, t'_n$ by considering successively the projections of t on the indices $e_1, \ldots, e_n$ belonging to the first, second, $\ldots, n^{\text{th}}$ dimension respectively. More precisely, for $k = 1, \ldots, n$, $t_k[x_1, \ldots, x_{n-1}]$ represents the $(n-1)$-dimensional array $t[x_1, \ldots, x_{k-1}, e_k, x_k, \ldots, x_{n-1}]$. The problem is to show that the arrays $t'_1, \ldots, t'_n$ have a nonempty intersection. Encoding the problem as is makes it simple to solve by our instantiation scheme, because the latter generates only one ground instance of each term. In order to make the problem more difficult, the multidimensional arrays are encoded as jagged arrays, i.e. as 1-dimensional arrays whose elements are arrays themselves. This is done inductively by inductively defining the terms s_i^k and t_j^k as

follows:

$$
\begin{aligned}
s_0^k &\stackrel{\text{def}}{=} t'_k, \\
s_{i+1}^k &\stackrel{\text{def}}{=} \texttt{select}(s_i^k, x_{i+1}) \quad \text{where } 0 \le i \le n-2; \\
t_0^k &\stackrel{\text{def}}{=} t, \\
t_{i+1}^k &\stackrel{\text{def}}{=} \texttt{select}(t_i^k, x_{i+1}) \quad \text{if } 0 \le i < k-1, \\
t_k^k &\stackrel{\text{def}}{=} \texttt{select}(t_{k-1}^k, e_k), \\
t_{i+1}^k &\stackrel{\text{def}}{=} \texttt{select}(t_i^k, x_i) \quad \text{if } k-1 < i \le n-1.
\end{aligned}
$$

The fact that t'_k is a projection on the kth dimension of array t is expressed by the formula $\forall x_1, \ldots, x_{n-1}. s_{n-1}^k \simeq t_n^k$, and the fact that the t'_ks have a nonempty intersection is formalized by:

$$\{\forall x_1, \ldots, x_{n-1}.\ s_{n-1}^k \simeq t_n^k \mid k \in [1..n]\} \models \exists y. \bigwedge_{i=1}^{n} \exists x_1, \ldots, x_{n-1}.\ s_{n-1}^i \simeq y.$$

For $n = 2$, we thus test the unsatisfiability of the following clause set, stating that any arbitrary row and column in a bidimensional array must intersect somewhere:

$$
\begin{aligned}
\forall z.\ \texttt{select}(t'_1, z) &\simeq \texttt{select}(\texttt{select}(t, e_1), z) \\
\forall w.\ \mathit{select}(t'_2, w) &\simeq \texttt{select}(\texttt{select}(t, w), e_2) \\
\forall x, y.\ \texttt{select}(t'_1, x) &\not\simeq \texttt{select}(t'_2, y)
\end{aligned}
$$

The results are depicted in the table below.

n	(I)+Yices
2	0.078
3	0.125
4	3.281
5	227.781

The number of instances increases exponentionally with n. Without using the instantiation scheme, Yices fails to detect the unsatisfiability of the formulas (i.e. it returns "unknown"), except for the trivial case $n = 1$. In fact, even when the problem is encoded using multidimensional arrays instead of jagged arrays, Yices does not detect the unsatisfiability of the formulas. This is not surprising since the quantifier elimination heuristics it uses are not complete and because, in contrast to the previous case, it is rather difficult to determine a priori (i.e. without constructing the proof) what the relevant instances are.

7 Discussion

In this paper we presented an instantiation scheme that permits to solve SMT problems in several theories of interest by simply testing the satisfiability of a set of ground clauses in first-order logic with equality. We also provided a set of syntactic conditions that guarantee the completeness of the scheme; these conditions have been implemented and were used to prove the scheme can be applied to the theories of Section 6. Note that these conditions do not assume that the set of axioms is saturated, although some *partial* saturation is sometimes required (see Section 5). The present paper is mostly focused on theoretical issues although some preliminary experiments are presented.

The scheme itself can be integrated into any SMT solver. Such an integration would permit to test which of the eager or lazy instantiation modes is the more efficient in practice. It will also be interesting to see whether model generation techniques can be applied to our setting, to generate relevant counter-examples for SMT problems.

There are some theories that are quite simple to define, but cannot be handled by our scheme. One such example is the theory of arrays, augmented with a constant predicate. This theory is defined by the axioms $(a_1), (a_2)$ of Section 6, along with axioms of the form

$$(\texttt{cst}) \ \ \neg\texttt{cst}(a, v) \ \vee \ \texttt{select}(a, x) \simeq v.$$

The predicate $\texttt{cst}(a, e)$ expresses the fact that the array a only contains the value e. This theory cannot be controlled. Indeed, since the equation $z \simeq w$ occurs in (a_2), w must occur in $I_{\mathcal{V}}(\texttt{select}(\texttt{store}(x, z, v), w) \simeq \texttt{select}(x, w))$, according to Definition 13. But in this case, index 2 must be in $\mathrm{I}_{vp}(\texttt{select})$, and the clause $\texttt{select}(a, x) \simeq v$ cannot be variable-preserving, since $x \in I_{\mathcal{V}}(\texttt{select}(a, x))$ but x does not occur in the term v. This theory is not $\mathcal{C}$-controllable either, since this would imply that $\texttt{select} \in \mathcal{C}$, so that (a_2) or $(\texttt{cst})$ can be reduced into a controlled clause by superposition from a $\mathcal{C}$-equation, (see Point 3 of Definition 21). But obviously, in this case new $\mathcal{C}$-equations can be derived (e.g. by superposition into the term $\texttt{store}(x, z, v)$ of (a_1)). These $\mathcal{C}$-equations are not strongly controlled by Definition, and this is explicitly forbidden by Point 4 of Definition 21. It turns out that the instantiation scheme is not complete for this theory. For example, the set of ground equations

$$\begin{array}{rc} 1: & \texttt{cst}(a, 0), \\ 2: & \texttt{cst}(b, 1), \\ 3: & b \simeq \texttt{store}(a, 0, 1), \\ 4: & 0 \not\simeq 1 \end{array}$$

is unsatisfiable. Indeed, clauses 1 and 2 imply that $\texttt{select}(a, 1) \simeq 0$, $\texttt{select}(b, 1) \simeq 1$. But since $b \simeq \texttt{store}(a, 0, 1)$ and $0 \not\simeq 1$ we have by (a_2): $\texttt{select}(b, 1) \simeq \texttt{select}(a, 1) \simeq 0 \not\simeq 1$. However, it is easy to check that the instantiation scheme cannot instantiate the variable w in (a_2) by 1, since 1 does not occur on the scope of a function symbol $\texttt{select}$. Thus the clause set obtained by instantiation is satisfiable. A closer inspection of this example shows that the theory of arrays augmented with a constant predicate can entail a finite cardinality clause, and our scheme does not handle these clauses, since they are trivially variable-eligible. We intend to investigate how such problematic cases can be avoided by applying the instantiation scheme to only a limited number of the clauses under consideration, in order to reason *modulo* the remaining clauses. This might allow us to handle specific theories such as bitvectors, or infinitely axiomatized theories, such as Presburger arithmetic or acyclic data structures.

Another direction to explore is how to raise the restriction that the instantiation rule only takes into account ground terms that occur in the original set of clauses. We intend to investigate how the heuristic techniques for quantifier reasoning may permit to consider new complex terms that can be used by the instantiation rule. Hopefully, taking these new terms into account will lead to a new scheme which will be incomplete, as it may falsely assert that a set is satisfiable, but will be able to perform more quantifier reasoning.

Another point that also deserves further investigation is the definition of an efficient algorithm for computing, for a given a clause set S, all the necessary parameters that make S controllable: selection function, ordering, set of function symbols $\mathcal{C}$ and index

functions I_0, I_{nv} and I_{vp}. Our current implementation assumes that all these parameters are given, and we intend to implement several optimizations in an upcoming version of our program. Actually, the difficult point is to compute the selection function (a crucial point, as evidenced by Example 13) and the set of symbols $\mathcal{C}$. Then the index functions may be easily derived from the definitions.

References

1. A. Abadi, A. Rabinovich, and M. Sagiv. Decidable fragments of many-sorted logic. *Journal of Symbolic Computation*, 45(2):153 – 172, 2010.
2. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 10(1):129–179, January 2009.
3. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
4. F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
5. L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.
6. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.
7. C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to sat. In E. Brinksma and K. G. Larsen, editors, *CAV 2002*, volume 2404 of *LNCS*, pages 236–249. Springer, 2002.
8. M. P. Bonacina and M. Echenim. On variable-inactivity and polynomial T-satisfiability procedures. *Journal of Logic and Computation*, 18(1):77–96, 2008.
9. M. P. Bonacina and M. Echenim. Theory decision by decomposition. *Journal of Symbolic Computation*, 45(2):229–260, 2010.
10. M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and undecidability results for Nelson-Oppen and rewrite-based decision procedures. In U. Furbach and N. Shankar, editors, *Proc. IJCAR-3*, volume 4130 of *LNAI*, pages 513–527. Springer, 2006.
11. A. R. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
12. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *Proc. VMCAI-7*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
13. M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communication of the ACM*, 5:394–397, 1962.
14. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960.
15. L. de Moura and N. Bjørner. Engineering DPLL(T) + saturation. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR-4*, volume 5195 of *LNAI*, pages 475–490. Springer, 2008.
16. L. M. de Moura and N. Bjørner. Efficient E-Matching for SMT Solvers. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
17. D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
18. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.
19. D. Dutertre and L. de Moura. The YICES SMT-solver, 2006. In SMT-COMP: Satisfiability Modulo Theories Competition. Available at http://yices.csl.sri.com.
20. M. Echenim and N. Peltier. Instantiation of SMT problems modulo Integers. In *AISC 2010 (10th International Conference on Artificial Intelligence and Symbolic Computation)*, LNCS. Springer, 2010.
21. H. Ganzinger and K. Korovin. Integrating equational reasoning into instantiation-based theorem proving. In *Computer Science Logic (CSL'04)*, volume 3210 of *LNCS*, pages 71–84. Springer, 2004.

22. Y. Ge, C. W. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):101–122, 2009.
23. 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.
24. S. Jacobs. Incremental instance generation in local reasoning. In F. Baader, S. Ghilardi, M. Hermann, U. Sattler, and V. Sofronie-Stokkermans, editors, *Notes 1st CEDAR Workshop, IJCAR 2008*, pages 47–62, 2008.
25. J. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Essays in Honor of Alan Robinson*, pages 91–99. The MIT-Press, 1991.
26. C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990.
27. H. Kirchner, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic combinability of rewriting-based satisfiability procedures. In M. Hermann and A. Voronkov, editors, *Proc. LPAR-13*, volume 4246 of *LNCS*, pages 542–556. Springer, 2006.
28. S. Lee and D. A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.
29. C. Lynch and D.-K. Tran. Automatic decidability and combinability revisited. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNAI*, pages 328–344. Springer, 2007.
30. L. Moura and N. Bjørner. Efficient e-matching for smt solvers. In *CADE-21: Proceedings of the 21st international conference on Automated Deduction*, pages 183–198, Berlin, Heidelberg, 2007. Springer-Verlag.
31. D. A. Plaisted and Y. Zhu. Ordered semantic hyperlinking. *Journal of Automated Reasoning*, 25(3):167–217, October 2000.
32. S. Ranise and D. Deharbe. Applying light-weight theorem proving to debugging and verifying pointer programs. In *Proc. of the 4th Workshop on First-Order Theorem Proving (FTP'03)*, volume 86 of *Electronic Notes in Theoretical Computer Science*, 2003.
33. U. Waldmann and V. Prevosto. SPASS+T. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proc. ESCoR Workshop, FLoc 2006*, volume 192 of *CEUR Workshop Proceedings*, pages 18–33, 2006.

The following technical appendices contain additional definitions and detailed proofs of important theorems used in the paper.

A Proof of Theorem 3

A.1 Derivation trees

Handling derivations in the usual sense is not really convenient for proving Theorem 3 because one has to apply complex transformations on derivations, involving for instance renamings of variables and propagations of instantiations. In order to properly handle these issues, we introduce the notion of *derivation trees*. These derivation trees allow us to keep track of the variables involved in derivations in a simple way.

We first fix a special deduction relation:

Definition 23 We denote by $\rightarrow^{\sigma}$ the relation $\rightarrow^{\sigma}_{sel,<}$, where $<$ denotes the smallest monotonic ordering such that $t < s$ if either t is a proper subterm of s or if $t \in \Sigma_0$ and $s \notin \Sigma_0$ and *sel* denotes the selection function defined as follows:

- $sel(C) = C$ if C is flat.
- Otherwise, $sel(C)$ is the set of literals in C that are either not flat or not ground[4].

This relation corresponds to a (partially) unrestricted version of the calculus (in the sense that the conditions on the inferences are as weak as possible), which is of course inefficient, but correct.

Definition 24 (Derivation tree) The class of *derivation trees* for a set of clauses S is the smallest set of expressions of the form $\tau = [C, \mathcal{T}, \sigma]$, such that:

[4] Obviously $sel(C)$ contains all the maximal literals in C.

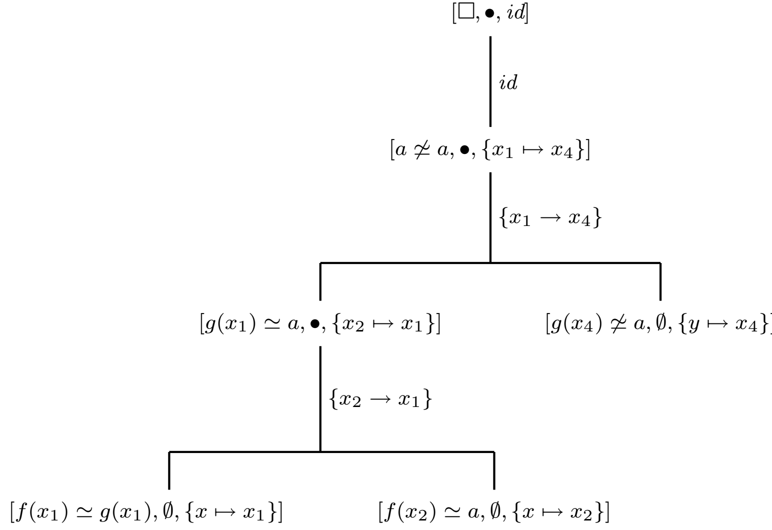

Fig. 5 Derivation tree of Example 15

- C is a clause, called the *root* of the tree and denoted by $\mathit{root}(\tau)$;
- $\mathcal{T}$ is a (possibly empty) set of pairwise variable-disjoint derivation trees for S;
- σ is a substitution;
- if $\mathcal{T} \neq \emptyset$ then $\mathit{root}(\mathcal{T}) \to^{\sigma} C$; otherwise, C is of the form $D\sigma$, where $D \in S$ and σ is a renaming.

The notation $\mathit{root}(\mathcal{T})$ is defined in a standard way as follows:

$$\mathit{root}(\mathcal{T}) \stackrel{\text{def}}{=} \{\mathit{root}(\tau) \mid \tau \in \mathcal{T}\}.$$

A derivation tree $[C, \mathcal{T}, \sigma]$ is *flat* if σ is flat and every derivation tree $\tau \in \mathcal{T}$ is flat. A *refutation tree* is a derivation tree of root $\Box$.

The set of clauses S may remain implicit, in which case we will simply talk about derivation trees.

Example 15 Let $S = \{f(x) \simeq g(x),\ f(x) \simeq a,\ g(y) \not\simeq a\}$. Then starting with the derivation trees

$$\tau_1 \ = \ [f(x_1) \simeq g(x_1), \emptyset, \{x \mapsto x_1\}] \ \text{ and } \ \tau_2 = [f(x_2) \simeq a, \emptyset, \{x \mapsto x_2\}],$$

we may construct the derivation tree $\tau_3 = [g(x_1) \simeq a, \{\tau_1, \tau_2\}, \sigma]$, where $\sigma = \{x_2 \to x_1\}$.

Consider $\tau_4 = [g(x_4) \not\simeq a, \emptyset, \{y \mapsto x_4\}]$ and $\sigma' = \{x_1 \to x_4\}$, then $\tau = [a \not\simeq a, \{\tau_3, \tau_4\}, \sigma']$ and $\tau' = [\Box, \{\tau\}, id]$ are derivation trees for S, which is unsatisfiable.

A graphical representation of τ' is provided in Figure 5. The black dots denote the derivation trees immediately below.

When clear from the context, we will keep the substitutions implicit in the graphical representations.

A.2 Restricted classes of derivation trees

We now restate the conditions on the derivations using the more precise formalism of derivation trees. The definition is actually slightly weaker than Definition 9 since the ordering and selection function are less restrictive.

Definition 25 (Simple Derivation Tree) Let S be a set of clauses. A derivation tree $\tau = [C, \mathcal{T}, \sigma]$ of S, where σ is an mgu of two terms t, s, is *simple* if it satisfies the following conditions:

1. σ is flat.
2. C is I_0-flat (see Section 2.3), and not variable-eligible (see Section 2.2).
3. If there is a position p such that $t|_p = f(t_1, \ldots, t_n)$ (resp. $s|_p = f(t_1, \ldots, t_n)$) and $t_i \in \mathcal{V}$ for some $i \in \mathrm{I}_{\mathrm{nv}}(f)$, then either $s|_p$ (resp. $t|_p$) occurs in S or $\sigma = id$.
4. If C is obtained using superposition/paramodulation by replacing a term $u\sigma$ by $v\sigma$, then $v\sigma \notin \mathcal{V}$.
5. Every derivation tree in $\mathcal{T}$ is simple.

A set of clauses S is *simply provable* if for all clauses C, if S admits a derivation tree of root C then S also admits a simple derivation tree of root C. A class of clause sets $\mathfrak{S}$ is *simply provable* if every set of clauses in $\mathfrak{S}$ is simply provable.

We first need to introduce additional definitions and results about derivation trees.

Definition 26 The *depth* of a derivation tree is inductively defined as follows:

$$depth([C, \emptyset, \sigma]) \stackrel{\text{def}}{=} 0, \quad \text{and} \quad depth([C, \mathcal{T}, \sigma]) \stackrel{\text{def}}{=} 1 + \max_{\tau \in \mathcal{T}} depth(\tau).$$

The set of variables occurring in a derivation tree τ, denoted by $V(\tau)$, is the set of variables defined as follows:

$$V([C, \mathcal{T}, \sigma]) \stackrel{\text{def}}{=} V(C) \cup \bigcup_{\tau' \in \mathcal{T}} V(\tau')$$

where $V(C)$ denotes the set of variables in C.

Example 16 In Example 15, the derivation tree τ' is of depth 3, and the set of variables occuring in τ' is $V(\tau') = \{x_1, x_2, x_4\}$.

We also define the composition of the unifiers over a derivation tree.

Definition 27 Given a derivation tree τ, the substitution μ_τ denotes the composition of the unifiers in τ, which is formally defined as follows:

$$\mu_{[C,\emptyset,\sigma]} \stackrel{\text{def}}{=} id,$$
$$\mu_{[C,\mathcal{T},\sigma]} \stackrel{\text{def}}{=} (\bigcup_{\tau \in \mathcal{T}} \mu_\tau)\sigma.$$

This notation is well-defined since the trees in $\mathcal{T}$ are mutually variable-disjoint. Note that the initial renamings of clauses are not taken into account in the construction of μ_τ. Thus, these substitutions only make sense when applied to the clauses of S that have already been renamed.

Example 17 In Example 15, we have $\sigma = \{x_2 \to x_1\}$ and $\sigma' = \{x_1 \to x_4\}$, hence $\mu_\tau = \sigma\sigma' = \{x_2 \to x_4, x_1 \to x_4\}$, and $\mu_{\tau'} = \mu_\tau$.

We obtain a first result for keeping track of a variable in a derivation tree.

Proposition 16 *Let $\tau = [C, \mathcal{T}, \sigma]$ be a derivation tree. If $x \in V(C)$ then $x \notin dom(\mu_\tau)$.*

Proof By an immediate induction on the depth of the tree. The variable x cannot occur in $dom(\sigma)$, since by definition of the calculus, σ is applied to the clauses in $\mathcal{T}$ to generated C, and is idempotent.

We define a subclass of simple derivation trees that will serve as a link between a derivation tree for S and one for $\hat{S}_\lambda$.

Definition 28 (Pure Derivation Trees) A derivation tree τ is *pure* if it is simple and if μ_τ is pure (this implies that every subtree of τ is pure).

We define the notions of instantiated and uninstantiated hypotheses in a derivation tree. These sets represent the clauses occurring in the original set of clauses, in their instantiated and uninstantiated versions.

Definition 29 Given a derivation tree τ, we denote by $hyp(\tau)$ the set of *uninstantiated hypotheses of* τ, and by $hyp^{\mathrm{inst}}(\tau)$ the set of *instantiated hypotheses of* τ. Formally:

$$hyp([C,\emptyset,\sigma]) \stackrel{\mathrm{def}}{=} hyp^{\mathrm{inst}}([C,\emptyset,\sigma]) \stackrel{\mathrm{def}}{=} \{C\sigma\},$$
$$hyp([C,\mathcal{T},\sigma]) \stackrel{\mathrm{def}}{=} \bigcup_{\tau\in\mathcal{T}} hyp(\tau),$$
$$hyp^{\mathrm{inst}}([C,\mathcal{T},\sigma]) \stackrel{\mathrm{def}}{=} \bigcup_{\tau\in\mathcal{T}} hyp^{\mathrm{inst}}(\tau)\sigma.$$

A clause C is a *main hypothesis* of a flat tree τ if $C \in hyp(\tau)$ and if for any variable $x \in V(C)$ and for any unifier σ occuring in τ, we have $x\sigma \in V(C) \cup \Sigma_0$.

Example 18 Suppose $\tau = [\square, \mathcal{T}, \{x \to a\}]$, where $\mathcal{T} = \{[p(x,b),\emptyset,id], [\neg p(a,y),\emptyset,id]\}$. Then $hyp(\tau) = \{p(x,b), \neg p(a,y)\}$ and $hyp^{\mathrm{inst}}(\tau) = \{p(a,b), \neg p(a,b)\}$.

It will sometimes be useful to replace a subtree in a simple or pure derivation tree by another one. This operation is harmless. Consider for example a derivation tree $\tau = [C,\mathcal{T},\sigma]$ that is simple (resp. pure), and let $\tau' \in \mathcal{T}$. Let τ'' be a simple (resp. pure) derivation tree with the same root as τ', and such that the only variables τ and τ'' have in common are those occurring in $\mathrm{root}(\tau'')$. Then $[C,\mathcal{T}',\sigma]$, where $\mathcal{T}' = (\mathcal{T} \setminus \{\tau'\}) \cup \{\tau''\}$, is a simple (resp. pure) derivation tree with the same root as τ. This result can be generalized to any subtree of τ:

Proposition 17 *Let τ be a simple (resp. pure) derivation tree for a set of clauses S, and $\tau' = [C,\mathcal{T},\sigma]$ be a subtree of τ. Let τ'' be a simple (resp. pure) derivation tree for S such that:*

- *τ' and τ'' have the same root,*
- *the only variables τ and τ'' have in common are those occurring in C.*

Then the derivation tree obtained by replacing τ' by τ'' in τ is also a simple (resp. pure) derivation tree for S, with the same root as τ.

If τ and τ'' have more variables in common than those in C, then the other variables can of course be renamed, thus yielding a derivation tree that satisfies all the conditions of Proposition 17.

Example 19 Consider the set of clauses

$$S = \{f(x) \simeq f'(x),\ f(x) \simeq g'(x),\ f'(x) \simeq h(x),\ h(x) \simeq g'(x),\ g(x) \simeq h(x)\}.$$

The derivation trees τ (top) and τ' (bottom) of Figure 6 have the same root and are both pure; τ' was obtained from τ by replacing the subtree of root $f(x_2) \simeq h(x_2)$, with the dashed lines, by the subtree with the bold lines. Note that these subtrees only have x_2 as a common variable.

We define a last class of derivation trees, the most constrained one, which permits only basic inferences to be carried out on a set of clauses. This class will allow us to discard the basic inferences in the construction of a derivation tree.

Definition 30 (Elementary Derivation Tree) A derivation tree $[C,\mathcal{T},\sigma]$ is *elementary* if $\mathcal{T} = \emptyset$ or $\mathcal{T} = \{\tau, \tau'\}$, where τ is elementary and τ' is pure and has a ground root.

Note that every elementary tree is pure. We introduce the following measure on derivation trees:

Definition 31 Given a derivation tree τ, the measure $\delta(\tau)$ is defined inductively by:

- $\delta(\tau) \stackrel{\mathrm{def}}{=} 0$ if τ is elementary.
- $\delta([C,\mathcal{T},\sigma]) \stackrel{\mathrm{def}}{=} 1 + \max_{\tau\in\mathcal{T}} \delta(\tau)$ otherwise.

In particular, this measure is unaffected by the adjunction of a flat ground clause to a hypothesis in a pure derivation trees:

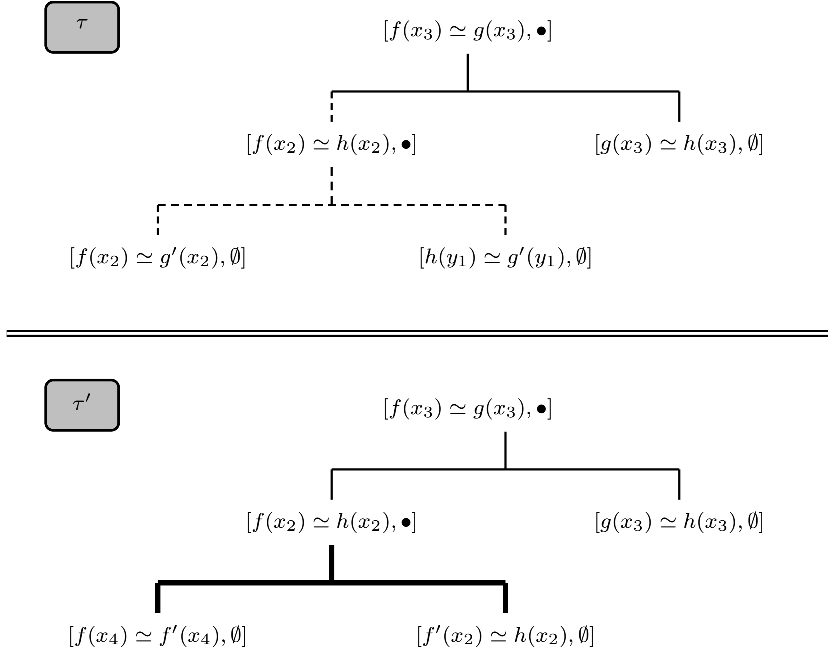

Fig. 6 Derivation trees of Example 19 (the substitutions are omitted for the sake of readability)

Proposition 18 *Let $\tau = [C, \mathcal{T}, \sigma]$ be a pure derivation tree for a set of clauses S, $D \in hyp(\tau)$, and consider a flat, ground clause E. Then there exists a pure derivation tree τ' of root $C \vee E$ for $S \cup \{D \vee E\}$, such that $\delta(\tau') = \delta(\tau)$.*

Proof We prove the result by induction on the depth of τ. If $\mathcal{T} = \emptyset$, then the result is obvious, since necessarily $C = D$. Now assume $\mathcal{T} = \{\tau_1, \tau_2\}$, where D_1, D_2 are the respective roots of τ_1, τ_2, and w.l.o.g., suppose $D \in hyp(\tau_1)$. Then by definition $\{D_1, D_2\} \to^{\sigma} C$, and by the induction hypothesis, there is a pure derivation tree τ_1' of root $D_1 \vee E$ for $S \cup \{D \vee E\}$. Since no ordering conditions are considered in the calculus, the literals in E have no influence on the derivation, and it is clear that $\{D_1 \vee E, D_2\} \to^{\sigma} C \vee E$. Thus, $\tau' = [C \vee E, \{\tau_1', \tau_2\}, \sigma]$ is a pure derivation tree for $S \cup \{D \vee E\}$, and obviously, $\delta(\tau') = \delta(\tau)$.

A.3 Swapping variables in pure derivation trees.

Given a pure derivation tree, we will sometimes construct a new derivation tree with the same properties as the original one, by swapping some of its variables, as in the following example.

Example 20 Let $S = \{f(x, y) \simeq g(x, y), f(y, y) \simeq h(y)\}$, and consider the clauses

$$\begin{aligned} C_1 &= f(x_1, x_2) \simeq g(x_1, x_2), \\ C_2 &= f(y_1, y_1) \simeq h(y_1), \\ C &= g(y_1, y_1) \simeq h(y_1), \end{aligned}$$

and the substitutions

$$\begin{aligned} \sigma_1 &= \{x \mapsto x_1,\ y \mapsto x_2\}, \\ \sigma_2 &= \{y \mapsto y_1\}. \end{aligned}$$

Let $\tau_1 = [C_1, \emptyset, \sigma_1]$, $\tau_2 = [C_2, \emptyset, \sigma_2]$, and $\sigma = \{x_1 \to y_1, x_2 \to y_1\}$; then $\tau = [C, \{\tau_1, \tau_2\}, \sigma]$ is a pure derivation tree for S.

Another pure derivation tree for S with root C can also be constructed by swapping x_1 and y_1. This tree is obtained by taking the clauses

$$C_1' = f(y_1, x_2) \simeq g(y_1, x_2),$$
$$C_2' = f(x_1, x_1) \simeq h(x_1),$$

the substitutions

$$\sigma_1' = \{x \mapsto y_1, y \mapsto x_2\},$$
$$\sigma_2' = \{y \mapsto x_1\},$$

and the derivation trees $\tau_1' = [C_1', \emptyset, \sigma_1']$ and $\tau_2' = [C_2', \emptyset, \sigma_2']$. Then $\tau' = [C, \{\tau_1', \tau_2'\}, \sigma]$ is also a pure derivation tree for S, and has the same root as τ.

To formalize the intuition of this example, we define the renaming of a derivation tree.

Definition 32 If $\tau = [C, \mathcal{T}, \sigma]$ is a derivation tree and π is a renaming, then $\tau\pi$ denotes the derivation tree defined as follows: $\tau\pi \stackrel{\text{def}}{=} [C\pi, \mathcal{T}\pi, \pi^{-1}\sigma\pi]$.

The next lemma shows that $\tau\pi$ is indeed a derivation tree with the same properties as τ. We prove the result in the case where τ is a pure derivation tree, along with some additional properties on its structure.

Lemma 14 *Let $\tau = [C, \mathcal{T}, \sigma]$ be a pure derivation tree for S and π be a renaming. Then $\tau\pi$ is a pure derivation tree τ' for S such that $root(\tau') = C\pi$, $\mu_{\tau'} = \pi^{-1}\mu_\tau\pi$, $hyp(\tau') = hyp(\tau)\pi$ and $\delta(\tau') = \delta(\tau)$.*

Proof The result is proven by induction on the depth of τ. If $\tau = [C, \emptyset, \sigma]$, then the result is obvious. Now assume $\tau = [C, \mathcal{T}, \sigma]$. Let $\mathcal{T} = \{\tau_1, \tau_2\}$ (we may have $\tau_1 = \tau_2$) and let $\tau_i' \stackrel{\text{def}}{=} \tau_i\pi$ $(i = 1, 2)$. By the induction hypothesis, τ_1' and τ_2' are derivation trees, and we have $root(\tau_i) = C_i\pi$, $\mu_{\tau_i'} = \pi^{-1}\mu_{\tau_i}\pi$, $hyp(\tau_i') = hyp(\tau_i)$, $\delta(\tau_i') = \delta(\tau_i)$ (for $i = 1, 2$). Then by definition, σ is the mgu of two terms s and t, so that $\sigma' = \pi^{-1}\sigma\pi$ is an mgu of $s\pi$ and $t\pi$. Thus, if $\mathcal{T}' = \{\tau_1', \tau_2'\}$, then $\tau' \stackrel{\text{def}}{=} [C\pi, \mathcal{T}', \sigma']$ is a pure derivation tree for S (since π is a renaming, τ_1' and τ_2' do not share any variables). Furthermore, by definition,

$$\begin{aligned}
\mu_{\tau'} &= (\bigcup_{\tau_i' \in \mathcal{T}'} \mu_{\tau_i'})\sigma' \\
&= (\bigcup_{\tau_i' \in \mathcal{T}'} \mu_{\tau_i'})\sigma' \\
&= (\bigcup_{\tau_i \in \mathcal{T}} \pi^{-1}\mu_{\tau_i}\pi)\sigma' \text{ (by the induction hypothesis)} \\
&= \pi^{-1}(\bigcup_{\tau_i \in \mathcal{T}} \mu_{\tau_i})\pi\sigma' \\
&= \pi^{-1}(\bigcup_{\tau_i \in \mathcal{T}} \mu_{\tau_i})\sigma\pi \text{ (because } \sigma' = \pi^{-1}\sigma\pi) \\
&= \pi^{-1}\mu_\tau\pi
\end{aligned}$$

It is simple to verify that $\delta(\tau') = \delta(\tau)$.

The previous lemma shows how pure derivation trees are preserved by renamings. In the sequel, we will sometimes need to rename variables in a pure derivation tree, without modifying the root of this derivation tree. The following lemma provides a sufficient condition for safely performing such an operation.

Lemma 15 *Let $\tau = [C, \mathcal{T}, \sigma]$ be a pure derivation tree for a set of clauses S, let $x \in dom(\sigma)$ and consider $\pi = \{x \to x\sigma, x\sigma \to x\}$. Then $\tau' = [C, \mathcal{T}\pi, \pi^{-1}\sigma]$ is a pure derivation tree for S such that $root(\tau') = C$, $\mu_{\tau'} = \pi^{-1}\mu_\tau$, $hyp(\tau') = hyp(\tau)\pi$ and $\delta(\tau') = \delta(\tau)$.*

Proof The proof is immediate if $\mathcal{T} = \emptyset$. By Lemma 14, $\tau'' = [C\pi, \mathcal{T}\pi, \pi^{-1}\sigma\pi]$ is a pure derivation tree for S of root $C\pi$, such that $hyp(\tau'') = hyp(\tau)\pi$, $\mu_{\tau''} = \pi^{-1}\mu_\tau\pi$, $\delta(\tau'') = \delta(\tau)$, where $\pi^{-1}\sigma\pi$ is an mgu of two terms $s\pi$ and $t\pi$. But since π is a renaming, $(\pi^{-1}\sigma\pi)\pi^{-1} = \pi^{-1}\sigma$ is also an mgu of $s\pi$ and $t\pi$, and the clause generated with this mgu is $(C\pi)\pi^{-1} = C$. Thus $\tau' = [C, \mathcal{T}\pi, \pi^{-1}\sigma]$ is a pure derivation tree for S. We have $\mu_{\tau'} = \mu_{\tau''}\pi^{-1} = \pi^{-1}\mu_\tau$.

In Example 20, C_1 and C_2 are hypotheses of τ, but none of them is a main hypothesis. But by swapping variables x_1 and y_1, the new derivation tree we obtain is such that C_1' is a main hypothesis for this tree. We show that it is possible to generalize this example.

Lemma 16 *Let τ be a pure derivation tree for a set of clauses S, let $D \in hyp(\tau)$ and let u denote a term appearing in D. If θ is a ground substitution such that $dom(\theta) \subseteq V(u)$, then there exists a pure derivation tree τ' for S with the same root as τ, a substitution η, a clause $D' = D\eta \in hyp(\tau')$ and a term u' occurring in D' such that:*

- *$D'\mu_{\tau'} = D\mu_\tau$ and $u'\mu_{\tau'} = u\mu_\tau$,*
- *D' a main hypothesis,*
- *the substitution $\theta' = \eta^{-1}\theta\eta$ corresponding to θ is such that $dom(\theta')\mu_{\tau'} \subseteq dom(\theta')$.*

Proof Assume that τ, D, u, θ do not satisfy the property above. In particular, $\eta \stackrel{\text{def}}{=} id$, $D' \stackrel{\text{def}}{=} D$ and $u' \stackrel{\text{def}}{=} u$ cannot be a solution, thus there must exist a variable x such that one of the following conditions holds:

- either $x \in V(D)$ and $x\mu_\tau \notin V(D)$ (i.e. D is not a main hypothesis of τ),
- or $x \in dom(\theta)$ and $x\mu_\tau \notin dom(\theta)$.

We denote by $E(\tau, D, u, \theta)$ the set of variables x satisfying one of these properties. By definition, for every $x \in E(\tau, D, u, \theta)$, we have $x \in dom(\mu_\tau)$, thus there exists a (unique) subtree $\tau_x = [C, \mathcal{T}, \sigma]$ of τ such that $x \in dom(\sigma)$. Let $m(\tau, D, u, \theta)$ denote the multiset defined by:

$$m(\tau, D, u, \theta) \stackrel{\text{def}}{=} \{\delta(\tau) - \delta(\tau_x) \mid x \in E(\tau, D, u, \theta)\}.$$

This measure $m(\tau, D, u, \theta)$ is clearly well-founded, thus we may assume w.l.o.g. that the tuple (τ, D, u, θ) is the minimal one (according to m) such that the above property does not hold.

Let $x \in E(\tau, D, u, \theta)$. Let $\tau_x = [C, \mathcal{T}, \sigma]$ and $x' = x\sigma$. We can safely replace τ_x by the pure derivation tree obtained as in Lemma 15. Let τ' be the pure derivation tree obtained by replacing τ_x with this new derivation tree in τ. Let $\pi = \{x \mapsto x', x' \mapsto x\}$ and $\theta' \stackrel{\text{def}}{=} \pi^{-1}\theta\pi$. By Lemma 15, $hyp(\tau')$ contains the clause $D' \stackrel{\text{def}}{=} D\pi$ and the term $u' \stackrel{\text{def}}{=} u\pi$. We have $D'\mu_{\tau'} = D\pi\mu_{\tau'}$. By Lemma 15, $\mu_{\tau'} = \pi^{-1}\mu_\tau$ thus $D'\mu_{\tau'} = D\mu_\tau$ and $u\mu_\tau = u'\mu_{\tau'}$.

By definition, we have $x \notin E(\tau', D', u', \theta')$, since x does not occur in D'. Let y be a variable distinct from x', occurring in $E(\tau', D', u', \theta')$. We show that $y \in E(\tau, D, u, \theta)$. Since τ' is obtained from τ by swapping x and x' in some part of the tree and since $x\sigma = x'\sigma = x'$, we have $y\mu_{\tau'} = y\mu_\tau$. We assume that $y \notin E(\tau, D, u, \theta)$ to derive a contradiction. We distinguish two possibilities:

- If $y \in V(D')$ and $y\mu_{\tau'} \notin V(D')$, then since $y \neq x, x'$ we have $y \in V(D)$ and since $y\mu_{\tau'} = y\mu_\tau$, $y\mu_\tau \notin V(D')$. If $y\mu_\tau \in V(D)$ this would imply that $y\mu_\tau = x$, which is impossible since $x\sigma = x'$.
- If $y \in dom(\theta')$ and $y\mu_{\tau'} \notin dom(\theta')$, then we have $y \in dom(\theta)$ (since $y \neq x, x'$) and $y\mu_\tau \notin dom(\theta')$. If $y\mu_\tau \in dom(\theta)$, then necessarily $y\mu_\tau = x'$, thus $x' \notin dom(\theta')$, and $x \notin dom(\theta)$. This means that both x and x' occur in D (x occurs in D since it is in $E(\tau, D, u, \theta)$ by definition and $x' \in dom(\theta) \subseteq V(D)$). Moreover since $y\mu_\tau = x'$ we have $x'\mu_\tau = x'$ and $x\mu_\tau = x'$. Consequently, x and $x\mu_\tau$ both occur in D and $x\mu_\tau \in dom(\theta)$. This contradicts the fact that $x \in E(\tau, D, u, \theta)$.

Consequently, the only variable that may occur in $E(\tau', D', u', \theta')$ but not in $E(\tau, D, u, \theta)$ is x'. If $x' \in E(\tau', D', u', \theta')$ then $\delta(\tau'_{x'}) > \delta(\tau_x)$. Thus $m(\tau', D', u', \theta') < m(\tau, D, u, \theta)$. Since τ is minimal, there exists a tree τ'' a clause D'' and a term u'' in D'' satisfying the desired properties for τ', D', u'. By transitivity, this also holds for τ, D, u.

Given a simple derivation tree τ of root C and a substitution θ, we provide a characterization based on the main hypotheses of τ that guarantees the existence of another simple derivation tree whose root is $C\theta$. We first prove a unification property that will be used to guarantee the existence of this simple derivation tree.

Lemma 17 *Let t, s be two unifiable terms and let $\sigma = mgu(t, s)$ be a flat substitution. Let θ be a flat and ground substitution such that $\forall x \in dom(\theta)$, $x\theta = x\sigma\theta$. Then $t\theta$ and $s\theta$ are unifiable. Furthermore, if $\eta = mgu(t\theta, s\theta)$, then $dom(\eta) = dom(\sigma) \setminus dom(\theta)$ and for all $x \in dom(\eta)$, $x\eta = x\sigma\theta$.*

Proof A *unification problem* is a conjunction of equations (or `false`). If ϕ, ψ are two unification problems, then we write $\phi \rightarrow_{unif} \psi$ if ψ is obtained from ϕ by applying the usual replacement or decomposition rules [25].

σ is of the form $\{x_i \mapsto u_i \mid i \in [1..m]\}$, where $x_1, \ldots, x_m$ are pairwise distinct variables not occurring in $u_1, \ldots, u_m$. Moreover, there exists a sequence of unification problems $(\phi_i)_{i\in[1..n]}$ such that $\phi_1 = (t =^? s)$, $\forall i \in [1..n-1], \phi_i \rightarrow_{unif} \phi_{i+1}$ and $\phi_n = \bigwedge_{i=1}^{m} (x_i =^? u_i)$. We assume that the unification rules are applied with the following priority: decomposition first, then replacement by terms w such that $w\theta \in T_0$ and finally the remaining replacements.

Let k be an index in $[1..n]$. We shall prove that $\phi_k\theta \rightarrow^*_{unif} \phi_{k+1}\theta$.

If ϕ_{k+1} is obtained from ϕ_k by decomposition, then ϕ_k, ϕ_{k+1} are respectively of the form $f(\mathbf{t}) =^? f(\mathbf{s}) \wedge \psi$ and $\mathbf{t} =^? \mathbf{s} \wedge \psi$. Then $\phi_k\theta$ is of the form $f(\mathbf{t})\theta =^? f(\mathbf{s})\theta \wedge \psi\theta$, thus the decomposition rule applies and deduces $\mathbf{t}\theta =^? \mathbf{s}\theta \wedge \psi\theta$ i.e. $\phi_{k+1}\theta$.

If ϕ_{k+1} is obtained from ϕ_k by replacement then ϕ_k, ϕ_{k+1} are respectively of the form $x =^? w \wedge \psi$ and $x =^? w \wedge \psi\{x \rightarrow w\}$ where x is a variable in $x_1, \ldots, x_m$ and w is a term not containing x. By definition, $x \in dom(\sigma)$. $\phi_k\theta$ is $x\theta =^? w\theta \wedge \psi\theta$.

If $x \notin dom(\theta)$ then $x\theta = x$. $x \notin V(w\theta)$ (since $x \notin V(w)$ and θ is ground). Thus the replacement rule applies and deduces $x =^? w\theta \wedge \psi\theta\{x \rightarrow w\theta\}$ i.e. $\phi_{k+1}\theta$.

Now assume that $x \in dom(\theta)$. We have, by definition, $x\sigma = w\sigma$, thus $x\sigma\theta = w\sigma\theta$ and by the above condition, since $x \in dom(\theta)$, $x\theta = w\sigma\theta$. If $w \in dom(\theta)$, or if $w \notin \mathcal{V}$ then we have $w\sigma\theta = w\theta$, thus $x\theta = w\theta$ and $\phi_k\theta = \phi_{k+1}\theta$. If $w \in \mathcal{V} \setminus dom(\theta)$ then we must have $x\sigma \neq w$ (otherwise we would have $w\theta = x\theta$ hence $w \in dom(\theta)$). This means that $w \in dom(\sigma)$ and w is replaced at some step. Thus there exists in the unification problem an equation of the form $y =^? x\sigma$, where $y \in \mathcal{V}$. But then, according to the above strategy, the replacement rule should have been applied first on this equation before $x =^? w$ (since $x\sigma\theta \in T_0$).

Therefore, we have $t\theta =^? s\theta \rightarrow^*_{unif} \bigwedge_{i=1}^{m} (x_i\theta =^? u_i\theta)$. By the above reasoning, for every $i, j \in [1..m]$, we have either $x_i\theta = x_i$ and $x_i \notin V(u_j\theta)$, or $x_i \in dom(\theta)$ and $x_i\theta = u_i\theta$. Thus $\{x_i \mapsto u_i\theta \mid i \in [1..m], x_i \notin dom(\theta)\} = mgu(t\theta, s\theta)$.

Lemma 18 *Let S be a set of clauses and $\tau = [C, \mathcal{T}, \sigma]$ be a simple derivation tree for S of root C. Consider a clause D that is a main hypothesis of τ and a flat and ground substitution θ such that $dom(\theta) \subseteq V(D)$ and for all variables $x \in dom(\theta)$, $x\theta = x\mu_\tau\theta$. Assume further that there exists a variable $x_0 \in V(D)$ such that $x_0\mu_\tau \in V(C)$. Then there exists a simple derivation tree τ' for $S \cup \{D\theta\}$, of root $C\theta$, such that $\delta(\tau') \leq \delta(\tau)$.*

Proof Let $S' = S \cup \{D\theta\}$, we will prove the result by induction on the depth of τ.

If $\mathcal{T} = \emptyset$, necessarily, $C = D$, and $C\theta \in S'$. Thus $\tau' \stackrel{\text{def}}{=} [C\theta, \emptyset, id]$ satisfies the requirements. Now assume that $\mathcal{T} = \{\tau_1, \tau_2\}$, where $root(\tau_1) = C_1 = (t_1 \simeq v) \vee C_1'$ and $root(\tau_2) = C_2[t_2]_p$, and suppose that C is generated by superposition or paramodulation from C_1 into C_2, i.e., $\sigma = mgu(t_1, t_2)$ and $C = (C_1' \vee C_2[v]_p)\sigma$. By definition, there exists an $i \in \{1, 2\}$ such that $D \in hyp(\tau_i)$. Moreover, $x_0\mu_{\tau_i}$ must occur in the root of τ_i, since $x_0\mu_\tau = x_0\mu_{\tau_i}\sigma \in V(C)$ by hypothesis, and $V(C) \subseteq V(C_1\sigma) \uplus V(C_2\sigma)$.

Let θ' be the restriction of $\sigma\theta$ to the variables in D; note that θ and θ' coincide on all the variables not occurring in $dom(\sigma)$. We check that we can apply the induction hypothesis on the derivation tree τ_i and the substitution θ'. First, it is clear that D is a main hypothesis of τ_i, since τ_i is a subtree of τ. Now let x be a variable occurring in $dom(\theta')$, we show that $x\theta' = x\mu_{\tau_i}\theta'$. If $x\mu_{\tau_i} = x$, then the proof is obvious. If $x\mu_{\tau_i}$ is a constant, then x cannot belong to $dom(\sigma)$ and of course, $x\mu_{\tau_i}\theta' = x\mu_{\tau_i}$. Since $x \in dom(\theta') \subseteq dom(\sigma) \cup dom(\theta)$, necessarily $x \in dom(\theta)$, and by hypothesis, $x\theta = x\mu_\tau\theta$. This entails that

$$x\theta' = x\sigma\theta = x\theta = x\mu_\tau\theta = x\mu_{\tau_i}\sigma\theta = x\mu_{\tau_i}.$$

Now assume $x\mu_{\tau_i}$ is a variable other than x, and let $y = x\mu_{\tau_i} \neq x$. Note that $y \in V(D)$ since D is a main hypothesis of τ_i. Since $y \neq x$, necessarily, $x \notin dom(\sigma)$ (by Proposition 16, because $x \in dom(\mu_{\tau_i})$). Furthermore, by definition, $x\mu_\tau = y\sigma$, thus:

$$x\theta' \ = \ x\sigma\theta \ = \ x\theta \ = \ x\mu_\tau\theta \ = \ y\sigma\theta \ = \ y\theta' \ = \ x\mu_{\tau_i}\theta'.$$

Therefore, we may apply the induction hypothesis on τ_i and θ': there exists a simple derivation tree τ_i' for S', of root $C_i\theta'$, where $\delta(\tau_i') \leq \delta(\tau_i)$.

Let $j = 3-i$, and let $\tau_j' = \tau_j\eta$ be a renaming of τ_j that contains no variable occurring in τ_i or τ_i'. Then of course, $C_j' = C_j\eta$ is the root of τ_j'. Since σ is a unifier of t_1, t_2 which are variable-disjoint, the substitution σ' such that $\forall x \in V(C_i), x\sigma' \stackrel{\text{def}}{=} x\sigma$ and $\forall x \in V(C_j'), x\sigma' \stackrel{\text{def}}{=} x\eta^{-1}\sigma$ is well-defined, flat, and is an mgu of $t_i, t_j\eta$. We now prove that $t_i\theta$ and $t_j\eta\theta$ are unifiable by verifying that the application conditions of Lemma 17 are satisfied for $t_i, t_j\eta$ and θ. By hypothesis, θ is flat and ground, and $\forall x \in dom(\theta), x\theta = x\mu_\tau\theta$. Furthermore, for all $x \in V(t_i)$, we have $x\mu_\tau = x\sigma$, and since σ and σ' are identical on $V(t_i)$, $x\mu_\tau = x\sigma'$. Thus, for all $x \in dom(\theta)$, $x\theta = x\sigma'\theta$. Consequently, we can apply Lemma 17: $t_i\theta$ and $t_j\eta\theta$ are unifiable, and have an m.g.u. γ such that $dom(\gamma) = dom(\sigma') \setminus dom(\theta)$ and $x\gamma = x\sigma'\theta$ for all $x \in dom(\gamma)$.

Since $dom(\theta) \subseteq V(D)$ and D is a main hypothesis, for all $x \in V(t_i) \cap dom(\theta)$, $x\sigma \in V(D)$ and $x\sigma\theta = x\theta'$. Hence,

$$t_j\eta\theta = t_j\eta, \quad \text{and}$$
$$t_i\theta = t_i\sigma\theta \ = \ t_i\theta'.$$

Therefore, γ is a flat, ground mgu of $t_i\theta'$ and $t_j\eta$, and the paramodulation rule is applicable on the clauses $C_i\theta'$ and $C_j\eta$.

We now prove that the generated clause is $C\theta$. To this aim, it suffices to show that $C_i\theta'\gamma = C_i\sigma\theta$ and $C_j\eta\gamma = C_j\sigma\theta$, since we use the unordered version of the calculus the inference step are stable by instantiation.

Let $x \in V(C_i)$. If $x \in V(D)$ then $x\theta' = x\sigma\theta$. By definition of γ, this implies that $x\theta'\gamma = x\sigma\theta$. If $x \notin V(D)$, then $x\theta' = x\theta = x$, hence $x\theta'\gamma = x\gamma = x\sigma'\theta'$. By definition of σ', we have $x\sigma' = x\sigma$. Furthemore, $x\sigma\theta' = x\sigma\theta$.

Now, let $x \in V(C_j)$. By definition $x \notin dom(\theta)$, hence $x\theta = x$. If $x\eta \notin dom(\sigma')$, then $x\eta\gamma = x$. Moreover, this implies that $x \notin dom(\sigma)$ thus $x\sigma\theta = x\theta = x$, and the proof is completed. Otherwise, by definition of γ, $x\eta\gamma = x\sigma\theta$.

Thus $\{C_i\theta', C_j\eta\} \rightarrow^\gamma C\theta$ and the derivation tree $\tau' = [C\theta, \{\tau_1', \tau_2'\}, \gamma]$ satisfies the desired result. This derivation tree is simple since it is obtained by instantiating simple derivation trees by a flat substitution (it is easy to verify that all the conditions in Definition 25 are satisfied).

If τ is nonelementary then $\delta(\tau) = 1 + \max(\delta(\tau_1), \delta(\tau_2)) \geq 1 + \max(\delta(\tau_1'), \delta(\tau_2')) = \delta(\tau')$. Otherwise, since the root of τ_i cannot be ground (since it contains the variable $x_0\mu_{\tau_i}$), τ_i must be elementary. Since $\delta(\tau_i') \leq \delta(\tau_i)$, necessarily, τ_i' is elementary. Furthermore, τ_j must be pure and have a ground root, hence, τ_j' is also pure and has a ground root. Therefore, τ' is elementary and $\delta(\tau) = \delta(\tau') = 0$.

The proof is similar if C is deduced using a unary inference step (reflection or equational factorisation).

A.4 Some properties of the relations $\equiv_C^S$

We state some properties of the relation $\equiv_C^S$ of Definition 2.

Definition 33 A set of clauses S is *Σ_0-stable* if for every pair of clauses $(C[a]_p, a \simeq b \vee D)$ occurring in S, the clause $C[b]_p \vee D$ also occurs in S.

Lemma 19 *Let S be a set of clauses and C be a flat ground clause. If S is Σ_0-stable then $\equiv_C^S$ is an equivalence relation.*

Proof $\equiv_C^S$ is reflexive by definition. Moreover it is symmetric by the commutativity of $\simeq$. We only have to show that it is transitive. Assume that $a \equiv_D^S c$ and $c \equiv_D^S b$, for some terms a, b, c. We show that $a \equiv_D^S b$. The proof is by induction on the depth of the terms. If $a = c$ or $c = b$ the proof is trivial. If $a = f(a_1, \dots, a_n)$ is complex, then by definition c must be of the form $f(c_1, \dots, c_n)$ where $\forall i \in [1..n].a_i \equiv_D^S c_i$. Similarly, we have $b = f(b_1, \dots, b_n)$ and

$\forall i \in [1..n].c_i \equiv^S_D b_i$. By the induction hypothesis, we have $\forall i \in [1..n].a_i \equiv^S_D b_i$ hence $a \equiv^S_D b$. If a is flat, then we must have $a, b, c \in \Sigma_0$ and S contains two clauses $(a \simeq c) \vee D_1$ and $(c \simeq b) \vee D_2$ where $D_1, D_2 \subseteq D$. By ground superposition (since the constant are unordered) we can derive $(a \simeq b) \vee D_1 \vee D_2$ from $(a \simeq c) \vee D_1$ and $(c \simeq b) \vee D_2$. Since S is Σ_0-stable, this clause must occur in S. Therefore, $a \equiv^S_C b$.

The relation $\succeq^S_C$ below provides a link between any complex term occurring in the root of a pure derivation tree, and the complex terms occurring in the original set of clauses.

Definition 34 Let S denote a set of clauses and C be a flat, ground clause.

Given two terms $t = f(t_1, \dots, t_n)$ and $s = f(s_1, \dots, s_n)$, we write $t \succeq^S_C s$ if for all $i \in [1..n]$, if $t_i \in T_0$ or $s_i \in \mathcal{V}$, then $t_i \equiv^S_C s_i$.

We prove some simple results on the relation $\succeq^S_C$: this relation is stable by instantiation and inclusion, and it is transitive.

Proposition 19 *Let C be a flat ground clause and S be a Σ_0-stable set of clauses. Then $\succeq^S_C$ is transitive.*

Proof Assume that $t \succeq^S_C s \succeq^S_C u$. By definition of $\succeq^S_C$ we have $t = f(t_1, \dots, t_n), s = f(s_1, \dots, s_n), u = f(u_1, \dots, u_n)$. Let $i \in [1..n]$. If $t_i \equiv^S_C s_i$ and $s_i \equiv^S_C u_i$ then $t_i \equiv^S_C u_i$ by transitivity of $\equiv^S_C$. If $t_i \equiv^S_C s_i$ and $s_i \notin T_0$ then by definition of $\equiv^S_C$ we must have $t_i = s_i$ thus $t_i \notin T_0$. If $t_i \notin T_0$ and $s_i \notin \mathcal{V}$ and $s_i \equiv^S_C u_i$ then we have either $s_i = u_i$ and the proof is obvious, of $s_i, u_i \in \Sigma_0$ thus $u_i \notin \mathcal{V}$.

Proposition 20 *Let t, s be terms, C be a flat ground clause, S be a set of clauses and σ be a flat substitution. If $t \succeq^S_C s$, then $t\sigma \succeq^S_C s\sigma$.*

Proof By definition, we have $t = f(t_1, \dots, t_n), s = f(s_1, \dots, s_n)$. Let $i \in [1..n]$. By definition of $\succeq^S_C$, one of the two following conditions holds:

- $t_i \equiv^S_C s_i$. By definition of $\equiv^S_C$, we have either $t_i = s_i$ and in this case $t_i\sigma = s_i\sigma$ or $t_i, s_i \in \Sigma_0$ hence $t_i\sigma = t_i, s_i\sigma = s_i$ and $t_i\sigma \equiv^S_C s_i\sigma$.
- $t_i \notin T_0$ and $s_i \notin \mathcal{V}$. Obviously $t_i\sigma$ cannot be flat and $s_i\sigma$ cannot be a variable.

Proposition 21 *Let t, s be terms, C, D be two ground flat clauses, S be a set of clauses. If $t \succeq^S_C s$ and $C \subseteq D$, then $t \succeq^S_D s$.*

Proof It suffices to remark that by definition of $\equiv^S_C$, we have: $u \equiv^S_C v \Rightarrow u \equiv^S_D v$. Then the result follows immediately by definition of $\succeq^S_C$.

Definition 35 Given a clause C, we denote by C^0 the disjunction of the literals in C that are both flat and ground.

Lemma 20 *Let S be a set of clauses and S' be the set of clauses that can be deduced from S using a pure derivation tree. If τ is a pure derivation tree of root C for S and t is a complex term appearing in C, then there exists a clause $C' \in hyp^{\mathrm{inst}}(\tau)$ containing a term s such that $s \succeq^{S'}_{C^0} t$.*

Proof The proof is by induction on the depth of τ. If $\tau = [C, \emptyset, id]$, then the result is obvious (by taking $C' = C$ and $s = t$). Now assume that $\tau = [C, \mathcal{T}, \sigma]$ and that $t = f(t_1, \dots, t_n)$ occurs in C. Note that since τ is pure, σ must be pure. By definition of the calculus, one of the following condition holds:

- either $t = t'\sigma$, where t' occurs in a clause $D \in root(\mathcal{T})$ (paramodulation "outside" of t', replacement by t', equational factoring or reflexivity rule),
- or $t = t'[v]_p\sigma$, where $p \neq \epsilon$ and t' occurs in a clause $D \in root(\mathcal{T})$ (paramodulation "inside" t').

In both cases, since σ is pure, t' must be of the form $f(t'_1, \ldots, t'_n)$. Consider the derivation tree $\tau' = [D, \mathcal{T}', \mu] \in \mathcal{T}$; since τ' is pure, by the induction hypothesis, $hyp^{\text{inst}}(\tau')$ contains a term $s = f(s_1, \ldots, s_n)$ such that $s \succeq^{S'}_{D^0} t'$. By definition, $s\sigma \in hyp^{\text{inst}}(\tau)$, and $s\sigma \succeq^{S'}_{D^0} t'\sigma$ by Proposition 20. The clause D cannot be flat since it contains s, thus $\text{sel}(D) \cap D^0 = \emptyset$ and the literals in D^0 are not affected by the inference step. This implies that $D^0 \subseteq C^0$, and by Proposition 21, $s\sigma \succeq^{S'}_{C^0} t'\sigma$.

In Case 1, $t = t'\sigma$ and the result is immediate. In Case 2, if position p is of length strictly greater than 1, then the flat arguments of $t'\sigma$ are not affected by the inference step and it is simple to check that $t'\sigma \succeq^{S'}_{C^0} t$; by Proposition 19, $s\sigma \succeq^{S'}_{C^0} t$ (since S' is obviously Σ_0-stable). Otherwise, $p = i$ for some $i \in [1..n]$, $root(\tau)$ contains a clause $(u \simeq v) \vee E$, and t is obtained from $f(t'_1, \ldots, t'_n)\sigma$ by replacing $t'_i\sigma = u\sigma$ by $v\sigma$. By Condition 4 in Definition 25, $v\sigma$ cannot be a variable.

- If $u\sigma$ is a complex term, then t'_i must also be a complex term since σ is flat. Since $t'_i \notin T_0$, s_i must also be complex, and the proof is complete.
- Otherwise, since u is eligible, u and v must be constant symbols, and E must be flat. Since $(u \simeq v) \vee E$ is not variable eligible, E must be ground. By definition of the calculus, $E \subseteq C^0$, hence $u \equiv^{S'}_{C^0} v$ and $s\sigma \succeq^{S'}_{C^0} t$.

Lemma 21 provides a link between the relations $\equiv^S_C$, $\succeq^S_C$, and the relation employed to define pseudo-unifiers (see Definition 6).

Lemma 21 *Let S be a Σ_0-stable set of clauses, C be a ground clause, and consider the terms s, t, u and v such that:*

- *$u \succeq^S_C t$ and $v \succeq^S_C s$,*
- *t and s are unifiable, with flat unifier σ.*

For all terms $s', t' \in T_0$ such that $s' \sim_{(u,v)} t'$, we have $s'\sigma \equiv^S_C t'\sigma$.

Proof By definition of $\succeq^S_C$, s, t, u and v are respectively of the form $f(s_1, \ldots, s_n), f(t_1, \ldots, t_n), f(u_1, \ldots, u_n)$ and $f(v_1, \ldots, v_n)$. We prove the result by induction on $\sim_{(u,v)}$.

- Suppose $t' = u_j$ and $s' = v_j$ for some $j \in [1..n]$. Then $u_j, v_j \in T_0$, and since σ is flat, $u_j\sigma, v_j\sigma \in T_0$. Since $u \succeq^S_C$ and $v \succeq^S_C$, by Proposition 20, $u\sigma \succeq^S_C t\sigma$ and $v\sigma \succeq^S_C s\sigma$, hence $u_j\sigma \equiv^S_C t_j\sigma$ and $v_j\sigma \equiv^S_C s_j$. Since $t_j\sigma = s_j\sigma$, we have the result by transitivity (Lemma 19).
- Suppose $t' \sim_{(u,v)} t''$ and $t'' \sim_{(u,v)}$ for some $t'' \in T_0$. Then by the induction hypothesis, $t'\sigma \equiv^S_C t''\sigma$ and $t''\sigma \equiv^S_C s'\sigma$; by transitivity, $t'\sigma \equiv^S_C s'\sigma$.

A.5 Completeness of the instantiation scheme

Now, we have all what we need to prove Theorem 3. Let $\tau = [C, \mathcal{T}, \sigma]$ be a simple derivation tree for S. Let S' be the set of clauses that can be deduced from S by a pure derivation tree. We shall prove by induction on $\delta(\tau)$ that there exists a pure derivation tree τ' for $\hat{S}$, with root C, and such that $\delta(\tau') \leq \delta(\tau)$. If τ is elementary then it is also pure, and there is nothing to prove. Otherwise, C must be deduced from (at most) two clauses D_1, D_2 (we may take $D_1 = D_2$ in case the rule is unary) and we have $\{D_1, D_2\} \to^\sigma C$. Moreover the two simple derivation trees τ_1, τ_2 for S, of respective roots D_1, D_2 are such that $\delta(\tau_i) < \delta(\tau)$ $(i = 1, 2)$.

By the induction hypothesis, there exist two pure derivation trees τ'_1 and τ'_2 of respective roots D_1 and D_2, such that $\delta(\tau'_i) \leq \delta(\tau_i)$ $(i = 1, 2)$. In particular, $D_1, D_2 \in S'$. By definition of the calculus, σ is the mgu of two terms s, t occurring in D_1, D_2 respectively. If σ is pure then the proof is obvious, since $[C, \{\tau'_1, \tau'_2\}, \sigma]$ is a pure derivation tree for $\hat{S}$ and $\delta([C, \{\tau'_1, \tau'_2\}, \sigma]) \leq \delta(\tau)$. We now assume that σ is not pure, which implies that there exists a position p such that exactly one of the two terms $t|_p, s|_p$ is a variable. We consider the case where $t|_p$ is a variable, the other case is symmetrical. W.l.o.g. we suppose that D_1, D_2 are the clauses with a minimal number of variables such that $\{D_1, D_2\} \to^\sigma C$ and there exist pure derivation trees τ'_1, τ'_2 for $\hat{S}$ of roots D_1, D_2 with $\delta(\tau'_i) \leq \delta(\tau_i)$ $(i = 1, 2)$.

The proof proceeds as follows. We show that p is of the form $q.i$, and by applying Lemma 20, we identify terms u', v' appearing in S such that $u'\mu_{\tau'_1} \succeq^{S'}_{C^0} t|_q$ and $v'\mu_{\tau'_2} \succeq^{S'}_{C^0} s|_q$. Then

we consider a substitution θ' based on the pseudo-unifier of u' and v', and apply Lemma 18 and the induction hypothesis to determine a pure derivation tree for $\hat{S}$ of root $D_1\theta' \vee E$, where $E \subseteq C^0$. Finally, we will see that $\{D_1\theta' \vee E, D_2\} \rightarrow^\sigma C$, thus exhibiting a contradiction with the fact that the number of variables in D_1 is minimal.

Determination of u' and v'. Since σ is flat, $s|_p$ must be a constant symbol. Furthermore, t, s cannot be variables, because D_1, D_2 are not variable-eligible (by Point 2 in Definition 25), and paramodulation into variables is forbidden by definition of the calculus. Thus $p = q.i$, and the terms $t|_q$ and $s|_q$ are of the form $f(t_1, \dots, t_m)$ and $f(s_1, \dots, s_m)$ respectively, with $t_i \in \mathcal{V}$ and $s_i \in \Sigma_0$.
By Lemma 20, $hyp^{\text{inst}}(\tau'_1)$ and $hyp^{\text{inst}}(\tau'_2)$ contain two clauses C_1, C_2 respectively containing terms of the form $u = f(u_1, \dots, u_m)$ and $v = f(v_1, \dots, v_m)$ such that $u \succeq^{S'}_{D_1^0} f(t_1, \dots, t_m)$ and $v \succeq^{S'}_{D_2^0} f(s_1, \dots, s_m)$. In particular, note that $u_i = t_i$. By definition of the selection function sel, since D_1, D_2 are not flat, $D_1^0 \cap sel(D_1) = D_2^0 \cap sel(D_2) = \emptyset$, and the literals in D_1^0 and D_2^0 are not affected by the inference step yielding C. Thus, $D_1^0, D_2^0 \subseteq C^0$, and by Proposition 21, we conclude that $u \succeq^{S'}_{C^0} f(t_1, \dots, t_m)$ and $v \succeq^{S'}_{C^0} f(s_1, \dots, s_m)$.
Since τ'_1 and τ'_2 are pure, there exist clauses C'_1 and C'_2 in $\hat{S}$ such that for $i \in \{1,2\}$, $C'_i\mu_i = C_i$, where $\mu_i = \mu_{\tau'_i}$ is a pure substitution; let $u' = f(u'_1, \dots, u'_m)$ and $v' = f(v'_1, \dots, v'_m)$ be the terms in C'_1, C'_2 such that $u'\mu_1 = u$ and $v'\mu_2 = v$, then $u'\mu_1 \succeq^{S'}_{C^0} t|_q$ and $v'\mu_2 \succeq^{S'}_{C^0} s|_q$. Let θ be the restriction of the pseudo-unifier of u' and v' to the variables in u', and θ' be the restriction of $\mu\sigma$ to $dom(\theta)$. By Lemma 16, up to swapping some variables in τ'_1, we may assume that C'_1 is a main hypothesis of τ'_1, and that $dom(\theta')\mu_{\tau'_1} \subseteq dom(\theta')$.

A pure derivation tree of root $D_1\theta'$. We assume that C'_1, C'_2 share no variable and denote by μ the substitution $\mu = \mu_{\tau'_1} \cup \mu_{\tau'_2}$. If $x \in dom(\theta)$, then $x\theta$ is a constant such that $x \sim_{(u',v')} x\theta$. Thus by Proposition 4, $x\mu \sim_{(u,v)} x\theta$ because μ is pure, and since S' is Σ_0-stable, by Lemma 21,

$$x\mu\sigma \ \equiv^{S'}_{C^0} \ x\theta. \qquad (\star)$$

By $(\star)$, $C'_1\theta'$ is obtained from $C'_1\theta$ by replacing some constant symbols $c_1, \dots, c_l$ by constant symbols $c'_1, \dots, c'_l$ such that $\forall j \in [1..l], c_j \equiv^{S'}_{C^0} c'_j$. By definition of $\equiv^{S'}_{C^0}$, for all $j \in [1..l]$, S' contains a clause of the form $c_j \simeq c'_j \vee E_j$, for some $E_j \subseteq C^0$. Clearly, there exists a flat ground clause $E \subseteq C^0$ such that the clause $C'_1\theta' \vee E$ is obtained by l applications of the (propositional, unordered) paramodulation rule into $C'_1\theta$ from $c_j \simeq c'_j \vee E_j$ $(1 \le j \le l)$. By definition of $\hat{S}$, since $C'_1, C'_2 \in \hat{S}$ and $\hat{S}$ is closed for the Instantiation rule, $C'_1\theta \vee E' \in \hat{S}$, where E' is the agreement condition of u' and v'. Thus S' also contains a clause $C'_1\theta' \vee E' \vee E$. By Lemma 21 for every disequation $c \not\simeq d$ occurring in E' we must have $c \equiv^{S'}_{C^0} d$ (since by definition $c \sim_{(u',v')} d$). But then the clause $C'_1\theta' \vee E$ must be also in S' (each disequation $c \not\simeq d$ can be deleted by a sequence of superposition steps followed by an application of the reflection rule).
Since τ'_1 is a pure derivation tree for S' of root D_1 and $C'_1 \in hyp(\tau'_1)$, by Proposition 18, there also exists a pure derivation tree for $S' \cup \{C'_1 \vee E\}$ of root $D_1 \vee E$, with the same measure $\delta(\tau'_1)$. Furthermore, since we assumed that C'_1 is a main hypothesis of τ'_1, we conclude that $C'_1 \vee E$ is a main hypothesis of this new tree. We now check that the application conditions of Lemma 18 are satisfied for $D_1 \vee E$, with substitution θ' and main hypothesis $C'_1 \vee E$. By definition, θ' is flat and ground, and $dom(\theta') \subseteq dom(\theta) \subseteq \mathcal{V}(C'_1 \vee E)$. Moreover, $C'_1 \vee E$ contains the variable u'_i, which is such that $u'_i\mu = u_i = t_i$ occurs in D_1. Now, let $x \in dom(\theta')$, we show that $x\mu_{\tau'_1}\theta' = x\theta'$. We assumed that $dom(\theta')\mu_{\tau'_1} \subseteq dom(\theta')$, which entails that $x\mu_{\tau'_1} = x\mu$ is also in $dom(\theta')$, thus, $x\mu_{\tau'_1}\theta' = x\mu\theta' = x\mu\mu\sigma = x\mu\sigma = x\theta'$ (because μ is idempotent).
Therefore, we may apply Lemma 18: there exists a simple derivation tree τ''_1 for $S' \cup \{C'_1\theta' \vee E\} = S'$ of root $D_1\theta' \vee E$, and such that $\delta(\tau''_1) \le \delta(\tau'_1) < \delta(\tau)$. By the induction hypothesis, $D_1\theta' \vee E$ admits a pure derivation tree τ_p for $\hat{S}$ such that $\delta(\tau_p) \le \delta(\tau''_1) < \delta(\tau)$.

Exhibiting the contradiction. Let x be a variable in D_1, then by Proposition 16, $x \notin dom(\mu)$ (since $\mu = \mu_{\tau_1} \cup \mu_{\tau_2}$). By definition, θ' is the restriction of $\mu\sigma$ to $dom(\theta)$; hence, if $x \in dom(\theta)$ then $x\theta' = x\mu\sigma = x\sigma$, and $x\theta'\sigma = x\sigma\sigma = x\sigma$, since σ is idempotent. If $x \notin dom(\theta)$ then $x\theta' = x$, and $x\theta'\sigma = x\sigma$. Thus $\{D_1\theta' \vee E, D_2\} \rightarrow^\sigma C$ (since $E \subseteq C^0$).

Since $v \succeq_{C^0}^{S'} f(s_1,\ldots,s_m)$, by definition, if $v_i \in T_0$, then $v_i \equiv_{C^0}^{S'} s_i$. Thus, if $v_i \in T_0$, then it must be a constant, since s_i is a constant. If $v_i \notin T_0$, then, since every clause occurring in τ is I_0-flat by Condition 2 of Definition 25, index i cannot be in $\mathrm{I}_0(f)$. Thus, this index is necessarily in $\mathrm{I}_{\mathrm{nv}}(f)$. Since $t_i \in \mathcal{V}$ and $\sigma \neq id$, by Condition 3 of Definition 25, s must occur in the initial clause set S, and we can safely replace C_2' with the clause containing s. Hence, we may assume w.l.o.g. that v_i is a constant, and therefore that v_i', which is such that $v_i = v_i'\mu_{\tau_2'}$, is also a constant.

Since $t_i \in \mathcal{V}$ and $u \succeq_{C^0}^{S'} f(t_1,\ldots,t_m)$, we have $u_i = t_i$, i.e., $u_i'\mu = t_i$. In particular, u_i' is a variable, and since v_i' is a constant, by definition of a pseudo-unifier, $u_i'\theta \in \Sigma_0$. Now, by $(\star)$, $u_i'\mu\sigma \equiv_{C^0}^{S'} u_i'\theta$, which means that $u_i'\mu\sigma = u_i'\theta'$ is a constant. Since μ is idempotent, $u_i'\theta' = u_i'\mu\sigma = u_i'\mu\mu\sigma = t_i\theta'$, hence $t_i\theta' \in \Sigma_0$. Consequently, $D_1\theta'$ is a strict instance of D_1, which contradicts the fact that the number of variables in D_1 is minimal.

B Proof of Lemma 4

We first show the following:

Lemma 22 *Let t be a term and σ be a flat substitution. Then $I_{\mathcal{V}}(t\sigma) = I_{\mathcal{V}}(t)\sigma \cap \mathcal{V}$.*

Proof We prove the result by induction on the depth of t. Note that t and $t\sigma$ must have the same depth, since σ is flat. If t is a variable or a constant, then by Definition 12, $I_{\mathcal{V}}(t) = \emptyset$. Moreover $t\sigma$ is also flat, thus $I_{\mathcal{V}}(t\sigma) = \emptyset$.

If $t = f(t_1,\ldots,t_n)$ then $I_{\mathcal{V}}(t) \stackrel{\text{def}}{=} \{t_i \mid i \in \mathrm{I}_{vp}(f), t_i \in \mathcal{V}\} \cup \bigcup_{i=1}^n I_{\mathcal{V}}(t_i)$, and $I_{\mathcal{V}}(t\sigma) \stackrel{\text{def}}{=} \{t_i\sigma \mid i \in \mathrm{I}_{vp}(f), t_i\sigma \in \mathcal{V}\} \cup \bigcup_{i=1}^n I_{\mathcal{V}}(t_i\sigma)$. If $t_i\sigma$ is a variable, then so is t_i, thus $\{t_i\sigma \mid i \in \mathrm{I}_{vp}(f), t_i\sigma \in \mathcal{V}\} = \{t_i \mid i \in \mathrm{I}_{vp}(f), t_i \in \mathcal{V}\}\sigma \cap \mathcal{V}$. Also, by the induction hypothesis, for $i \in [1..n]$, $I_{\mathcal{V}}(t_i\sigma) = I_{\mathcal{V}}(t_i)\sigma \cap \mathcal{V}$. Consequently $I_{\mathcal{V}}(t\sigma) = I_{\mathcal{V}}(t)\sigma \cap \mathcal{V}$.

Let C be a variable-preserving clause and let σ be a flat substitution. We have to show that for all $L \in C$, $L\sigma$ is variable-preserving in $C\sigma$. Let $C = L \vee D$. Since C is variable-preserving, L is variable-preserving in C, thus one of the following conditions holds:

- L is of the form $t \not\simeq x$ (or $x \not\simeq t$), where $x \in I_{\mathcal{V}}(t)$. Since σ is flat, $x\sigma$ is either a variable or a constant symbol. If $x\sigma \in \mathcal{V}$ then by Lemma 22, $x\sigma \in I_{\mathcal{V}}(t\sigma)$ thus $t\sigma \not\simeq x\sigma$ is variable-preserving in $C\sigma$. Otherwise $x\sigma$ is ground hence $I_{\mathcal{V}}(x\sigma) = \emptyset$ and $t\sigma \not\simeq x\sigma$ is also variable-preserving by Condition (1b).
- L is of the form $t \not\simeq s$, where $I_{\mathcal{V}}(t) = \emptyset$ or $I_{\mathcal{V}}(s) = \emptyset$. Say $I_{\mathcal{V}}(t) = \emptyset$. Then by Lemma 22 we have $I_{\mathcal{V}}(t\sigma) = \emptyset$, hence $L\sigma$ is variable-preserving in $C\sigma$ by Condition (1b).
- L is of the form $t \not\simeq s$, where $I_{\mathcal{V}}(t) \cup I_{\mathcal{V}}(s) \subseteq I_{\mathcal{V}}(D)$. Let y be a variable in $I_{\mathcal{V}}(t\sigma) \cup I_{\mathcal{V}}(s\sigma)$. By Lemma 22, $y = x\sigma$ for some $x \in I_{\mathcal{V}}(t) \cup I_{\mathcal{V}}(s)$. Then $x \in I_{\mathcal{V}}(D)$ by hypothesis, and by Lemma 22, $x\sigma = y \in I_{\mathcal{V}}(D\sigma)$. Thus Condition (1c) holds and $L\sigma$ is variable-preserving in $C\sigma$.
- L is of the form $t \simeq s$, where $t, s \notin \mathcal{V}$ and $I_{\mathcal{V}}(t) = I_{\mathcal{V}}(s)$. By Corollary 2, $I_{\mathcal{V}}(t\sigma) = I_{\mathcal{V}}(s\sigma)$. Moreover since $t, s \notin \mathcal{V}$, necessarily, $t\sigma, s\sigma \notin \mathcal{V}$ Therefore $L\sigma$ is variable-preserving in $C\sigma$.
- If L is of the form $t \simeq s$, where $t, s \in T_0$ and $\{t, s\} \cap \mathcal{V} \subseteq I_{\mathcal{V}}(D)$. Then since σ is flat, $t\sigma, s\sigma \in T_0$. Let $x \in \{t\sigma, s\sigma\} \cap \mathcal{V}$. Obviously, x is of the form $y\sigma$ for some $y \in \{t, s\}$, and $y \in I_{\mathcal{V}}(D)$ by hypothesis. Hence, $x \in I_{\mathcal{V}}(D\sigma)$ by Lemma 22.

C Proof of Theorem 4

We need some preliminary results.

Proposition 22 *If s is a subterm of t, then $I_{\mathcal{V}}(s) \subseteq I_{\mathcal{V}}(t)$.*

We obtain as a simple consequence:

Corollary 2 *Let t, s be two terms such that $I_{\mathcal{V}}(t) = I_{\mathcal{V}}(s)$ and let σ be a flat substitution. Then $I_{\mathcal{V}}(t\sigma) = I_{\mathcal{V}}(s\sigma)$.*

The following lemmata provide exhibit conditions that guarantee variable-preservation is maintained by operations such as disjunctions, instantiations and replacements.

Definition 36 A literal L is *dominated* in a clause $L \vee C$ if $I_{\mathcal{V}}(L) \subseteq I_{\mathcal{V}}(C)$.

Lemma 23 *If L is dominated in a clause $L \vee C$ and $L \vee C$ is variable-preserving, then C is variable-preserving.*

Proof Assume that C is not variable-preserving. Then there exists a literal $L' \in C$ that is not variable-preserving in C. Since L' is variable-preserving in $L \vee C$, this implies by Definition 13 that one of Conditions (1c) or (2b) does not hold for the clause C. Hence, there exists a variable x occurring in $I_{\mathcal{V}}(L)$ but not in $I_{\mathcal{V}}(C)$. But this is impossible since L is dominated.

We show that the sets $I_{\mathcal{V}}(t)$ are stable by replacement.

Lemma 24 *Let t, s be two terms, p be a position in t, and assume that $t|_p$ and s are not variables. If $I_{\mathcal{V}}(s) = I_{\mathcal{V}}(t|_p)$ then $I_{\mathcal{V}}(t[s]_p) = I_{\mathcal{V}}(t)$.*

Proof We prove the result by induction on the length of p. Note that since $t|_p$ is not a variable, t cannot be a variable.

If $p = \epsilon$, then $t|_p = t$ and $t[s]_p = s$. Thus we have $I_{\mathcal{V}}(t) = I_{\mathcal{V}}(t[s]_p) = I_{\mathcal{V}}(s)$.

If $p = i.q$, then t is of the form $f(t_1, \ldots, t_n)$, thus $t|_p = t_i|_q$ and $t[s]_p = f(t_1, \ldots, t_{i-1}, t_i[s]_q, t_{i+1}, \ldots, t_n)$. Since $t|_p$ and s are not variables, t_i and $t_i[s]_q$ cannot be variables. By the induction hypothesis $I_{\mathcal{V}}(t_i[s]_q) = I_{\mathcal{V}}(t_i)$, hence by Definition 12,

$$\begin{aligned} I_{\mathcal{V}}(t[s]_p) &= \{t_j \mid j \in \mathrm{I}_{vp}(f), t_j \in \mathcal{V}\} \cup \bigcup_{j \in [1..n] \setminus \{i\}} I_{\mathcal{V}}(t_j) \cup I_{\mathcal{V}}(t_i[s]_p) \\ &= \{t_j \mid j \in \mathrm{I}_{vp}(f), t_j \in \mathcal{V}\} \cup \bigcup_{j \in [1..n]} I_{\mathcal{V}}(t_j) \\ &= I_{\mathcal{V}}(t). \end{aligned}$$

Lemma 24 implies that variable-preserving clauses are also stable by replacement:

Lemma 25 *Let C be a variable-preserving clause, $u = C|_p$ be a non-variable term occurring in C and v be a non-variable term such that:*

- $I_{\mathcal{V}}(u) = I_{\mathcal{V}}(v)$,
- *if* $u \in \Sigma_0$ *then* $v \in \Sigma_0$.

Then $C[v]_p$ is variable-preserving.

Proof Let L be the literal containing the term t such that $u = t|_q$ for some position q. By Lemma 24, $I_{\mathcal{V}}(t) = I_{\mathcal{V}}(t[v]_p)$, and since u and v are not variables, it is simple to check that if L satisfies one of Conditions (1a)-(2a), then so does the literal obtained after replacing u by v. If L satisfies Condition (2b), then necessarily u is a constant, hence by hypothesis, so is v. Again, the literal obtained after replacing u by v satisfies Condition (2b).

We are now in position to give the proof of Theorem 4. We distinguish several cases according to the rule used to derive C. Note that in all cases, by Proposition 6, none of the premisses of the inference step are variable-eligible.

C is generated by the superposition or paramodulation rule. This means that C is of the form $D_1[v]_p\sigma \vee D'_2\sigma$, where D_1 and $D_2 = u \simeq v \vee D'_2$ are clauses in S', $t = D_1|_p$, and $\sigma = mgu(t, u)$. Since D_1 and D_2 are variable-preserving, by Lemma 4, $D_1\sigma$ and $D_2\sigma$ are also variable-preserving. Since there is no superposition/paramodulation into variables, $D_1|_p \notin \mathcal{V}$, and since $u \simeq v \vee D'_2$ is not variable-eligible, $u \notin \mathcal{V}$. Since D_2 is variable-preserving, one of Conditions (2a) or (2b) must hold for literal $u \simeq v$.

- If Condition (2a) holds, then $I_{\mathcal{V}}(u) = I_{\mathcal{V}}(v)$, and v cannot be a variable. By definition of the ordering, if $u\sigma$ is a constant, then $v\sigma$ must also be a constant (since constants are strictly smaller than complex terms).
- If Condition (2b) holds, then u is necessarily a constant. If v were a variable, then we would have $v \in I_{\mathcal{V}}(D')$. Thus, D' would contain a term $f(t_1, \ldots, t_n)$ of which v is a subterm, and we would have $u < f(t_1, \ldots, t_n)$, contradicting the fact that $u \simeq v$ is a maximal literal in D_2. Thus, v must be a constant, and $I_{\mathcal{V}}(u) = I_{\mathcal{V}}(v) = \emptyset$.

By Lemma 4, $D_1\sigma$ is variable-preserving. Since σ is an mgu of u and t, we have $I_{\mathcal{V}}(t\sigma) = I_{\mathcal{V}}(u\sigma)$, and by Corollary 2, $I_{\mathcal{V}}(t\sigma) = I_{\mathcal{V}}(v\sigma)$. Therefore, by Lemma 25, $D_1[v]_p\sigma$ is variable-preserving, and by Proposition 5, $D_1[v]_p\sigma \vee D_2\sigma$ is also variable-preserving.

Since $I_{\mathcal{V}}(u\sigma) = I_{\mathcal{V}}(v\sigma)$ and $v\sigma$ occurs in $D_1[v]_p\sigma$, by Proposition 22, $(u \simeq v)\sigma$ is dominated in $D_1[v]_p\sigma \vee D_2\sigma$. By Lemma 23, $D_1[v]_p\sigma \vee D_2'\sigma$ is variable-preserving.

C is generated by the reflection rule. C is of the form $D\sigma$, where S' contains a clause of the form $t \not\simeq s \vee D$ and $\sigma = mgu(t, s)$. By Lemma 4, $(t \not\simeq s \vee D)\sigma$ is variable-preserving, we show that $t\sigma \not\simeq s\sigma$ is dominated in $(t \not\simeq s \vee D)\sigma$.

Let $x \in I_{\mathcal{V}}(t\sigma) \cup I_{\mathcal{V}}(s\sigma)$; by Lemma 22, $x = y\sigma$, where $y \in I_{\mathcal{V}}(t) \cup I_{\mathcal{V}}(s)$. Since $t \not\simeq s$ is variable-preserving in $t \not\simeq s \vee D$, one of Conditions (1a)-(1c) must hold.

- If Condition (1a) holds, then we have, say, $s \in I_{\mathcal{V}}(t)$. By Definition 12, this implies that s is a strict subterm of t, which is impossible since t, s are unifiable.
- If Condition (1b) holds, then $I_{\mathcal{V}}(t) = \emptyset$ or $I_{\mathcal{V}}(s) = \emptyset$, say $I_{\mathcal{V}}(t) = \emptyset$. By Lemma 22 $I_{\mathcal{V}}(t\sigma) = \emptyset$, and since $t\sigma = s\sigma$, we conclude that $I_{\mathcal{V}}(s\sigma) = \emptyset$. This contradicts the fact that $x \in I_{\mathcal{V}}(t\sigma) \cup I_{\mathcal{V}}(s\sigma)$.
- If Condition (1c) holds, then $y \in I_{\mathcal{V}}(L')$, for some $L' \in D$. But in this case, by Lemma 22, $x = y\sigma \in I_{\mathcal{V}}(L'\sigma) \subseteq I_{\mathcal{V}}(D\sigma)$.

Thus $t\sigma \not\simeq s\sigma$ is dominated in $(t \not\simeq s \vee D)\sigma$ and by Lemma 23, $D\sigma$ is variable-preserving.

C is generated by the equational factorisation rule. C is of the form $(D \vee s \not\simeq v \vee t \simeq s)\sigma$, where S' contains a clause of the form $(D \vee u \simeq v \vee t \simeq s)$, and $\sigma = mgu(u, t)$.

By definition, since $D \vee u \simeq v \vee t \simeq s$ is variable-preserving, we have $I_{\mathcal{V}}(t) = I_{\mathcal{V}}(s)$ and $I_{\mathcal{V}}(u) = I_{\mathcal{V}}(v)$, regardless of which of Conditions (2a) or (2b) holds. Thus $I_{\mathcal{V}}(v\sigma) = I_{\mathcal{V}}(u\sigma)$ by Corollary 2, and since σ is a unifier of u and t, we deduce that $I_{\mathcal{V}}(v\sigma) = I_{\mathcal{V}}(t\sigma)$ and $I_{\mathcal{V}}(s\sigma) \cup I_{\mathcal{V}}(v\sigma) \subseteq I_{\mathcal{V}}(s\sigma) \cup I_{\mathcal{V}}(t\sigma)$. Therefore, $(s \not\simeq v)\sigma$ satisfies Condition (1c), and is variable-preserving in C.

We now show that $(D \vee t \simeq s)\sigma$ is variable-preserving. By Lemma 4, $(D \vee u \simeq v \vee t \simeq s)\sigma$ is variable-preserving. But since $I_{\mathcal{V}}(u\sigma) = I_{\mathcal{V}}(v\sigma)$ and $u\sigma = t\sigma$, we conclude that $I_{\mathcal{V}}((u \simeq v)\sigma) \subseteq I_{\mathcal{V}}((s \simeq t)\sigma)$, which means that $(u \simeq v)\sigma$ is dominated in $(D \vee u \simeq v \vee t \simeq s)\sigma$, thus $(D \vee t \simeq s)\sigma$ is variable-preserving, by Lemma 23. We conclude that C is variable-preserving.

D Input File

```
/***************************************************************************/
/* SMT */
/***************************************************************************/

/***************************************************************************/
/* function declaration */
/***************************************************************************/

/* nonvar, flat and preserving respectively denote the indexes
   in IO, Inv and Inst */

function select : [nonvar = [1], flat = [2], preserving = [2] ].
function ord : [flat = [1,2]].
function inf : [nonvar = [1,2]].
function car : [nonvar = [1] ].
function cdr : [nonvar = [1] ].
function prev : [nonvar = [1] ].
function next : [nonvar = [1] ].
function cons : [nonvar = [1,2]].
function enc : [nonvar = [1,2]].
function dec : [nonvar = [1,2]].
function s : [nonvar = [1]].
function p : [nonvar = [1]].
function succ : [nonvar = [1]].
function ia : [nonvar = [1]].
function sa : [nonvar = [1]].
function store : [nonvar = [1,2,3]].
```

```
%constants = [nil, true]. % not necessary

/* The following list contains all the symbols in C */

restricted_symbols = [enc,dec,s,succ,p,sa,ia,eqa,partitioned,bsorteda,
beqa,cons,car,cdr,store,next,prev].

/*************************************************************************/
/* Theories */
/*************************************************************************/

/* The syntax is straightforward. */

/* <clause> = <cl_id> : [ <lit_list> ] | [] */
/* <lit_list> = <literal> {, literal}* */
/* <literal> = <atom> | not(<atom>) */
/* <atom> = <term> = <term> */
/* terms are written using Prolog conventions, in particular variables
   start with a capital letter */

/* sets of clauses can be constructed from cl_id or other clause sets */

/* ordering */

o1 : [not(ord(X,Y) = true), not(ord(Y,X) = true)].
o2 : [not(ord(X,Y) = true), not(ord(Y,Z) = true), ord(X,Z) = true].

ord = [o1,o2].

/* natural number */

n1 : [not(0 = succ(_Y))].
n2 : [X = Y, not(succ(X) = succ(Y))].

nat = [n1,n2].

/* inf on nat */

inf1 : [not(inf(X,Y) = true), not(inf(succ(X),succ(Y))= true)].
inf2 : [inf(0,succ(_X)) = true].

inf = [inf1,inf2].

/* integer offset */

i1 : [p(s(X)) = X].
i2 : [s(p(X)) = X].
i3 : [not(s(X) = X)].
i4 : [not(p(X) = X)].

integeroffset = [i1,i2,i3,i4].

/* list */

l1 : [car(cons(X,_Y)) = X].
l2 : [cdr(cons(_X,Y)) = Y].
l3 : [X = nil, cons(car(X),cdr(X)) = X].
l4 : [not(cons(X,Y) = nil)].

list = [l1,l2,l3,l4].
```

```
/* encryption */

e1 : [enc(dec(X,Y),Y) = X].
e2 : [dec(enc(X,Y),Y) = X].

encrypt = [e1,e2].

/* linked list */

ll1 : [X = nil, next(X) = nil, prev(next(X)) = X].
ll2 : [X = nil, prev(X) = nil, next(prev(X)) = X].
ll3 : [not(prev(X) = prev(Y)), prev(X) = nil, prev(Y) = nil, X = Y, X = nil, Y = nil].
ll4 : [not(next(X) = next(Y)), next(X) = nil, next(Y) = nil, X = Y, X = nil, Y = nil].

ll = [ll1,ll2,ll3,ll4].

/* array */

array1 : [select(store(_T,I,V),I) = V].
array2 : [I = J, select(store(T,I,_V),J) = select(T,J)].
array3 : [not(sa(T) = true), select(select(T,I),J) = select(select(T,J),I)].
array4 : [not(ia(T) = true), I = J, not(select(T,I) = select(T,J))].
array5 : [not(eqa(T,S) = true), select(T,I) = select(S,I)].
array6 : [not(beqa(T,S,L,U) = true), not(inf(L,I) = true),
not(inf(I,U) = true), select(T,I) = select(S,I)].
array7 : [not(bsorteda(T,L,U) = true), not(inf(L,I) = true), not(inf(I,U) = true),
inf(select(T,I),select(S,I)) = true].
array8 : [not(bsorteda(T,L,U) = true), not(inf(L,I) = true), not(inf(I,J) = true),
not(inf(J,U) = true),inf(select(T,I),select(S,J)) = true].
array9 : [not(partitioned(T,L1,U1,L2,U2) = true), not(inf(L1,I) = true),
not(inf(I,U1) = true), not(inf(U1,L2) = true),not(inf(L2,J) = true), not(inf(J,U2) = true),
 inf(select(T,I),select(S,J)) = true].

array = [array1,array2,array3,array4,array5,array6,array7,array8,array9].

/****************************************************************************/

all = [ord,nat,integeroffset,inf,array,ll,encrypt,list].

/****************************************************************************/
/* Specify the negative literal that should be selected */
/* (ordering is built-in LPO) */

select_lit not(sa(T) = true).
select_lit not(ia(T) = true).
select_lit not(eqa(T,S) = true).
select_lit not(beqa(T,S,L,U) = true).
select_lit not(bsorteda(T,L,U) = true).
select_lit not(partitioned(T,L,U,LL,UU) = true).
```