A rule-based system for automatic decidability and combinability

Elena Tushkanova, Alain Giorgetti, Christophe Ringeissen, Olga Kouchnarenko

A Rule-Based System for Automatic Decidability and Combinability

Elena Tushkanova*[1,2], Alain Giorgetti†[1,2], Christophe Ringeissen‡[1], Olga Kouchnarenko§[1,2]

[1]Inria, Villers-les-Nancy, F-54600, France
[2]FEMTO-ST (UMR 6174), Univ. of Franche-Comté, Besançon, F-25030, France

Abstract

This paper deals with decision procedures specified by using a superposition calculus which is an inference system at the core of all equational theorem provers. This calculus is refutation complete: it provides a semi-decision procedure that halts on unsatisfiable inputs but may diverge on satisfiable ones. Fortunately, it may also terminate for some theories of interest in verification, and thus it becomes a decision procedure. To reason on the superposition calculus, a schematic superposition calculus has been developed to build the schematic form of the saturations allowing to automatically prove decidability of single theories and of their combinations.

This paper presents a rule-based logical framework and a tool implementing a complete many-sorted schematic superposition calculus for arbitrary theories. By providing results for unit theories, arbitrary theories, and also for theories with counting operators, we show that this tool is very useful to derive decidability and combinability of theories of practical interest in verification.

1 Introduction

Satisfiability procedures modulo background theories such as classical data structures (e.g., lists, records, arrays, ...) are at the core of many state-of-the-art verification tools. Designing and implementing satisfiability procedures is a very complex task, where one of the main difficulties consists in proving their soundness.

To overcome this problem, the rewriting approach [1] allows building satisfiability procedures in a flexible way, by using a superposition calculus [2] (also called *Paramodulation Calculus* in [3]). In general, a fair and exhaustive application of the rules of this calculus leads to a semi-decision procedure that halts on unsatisfiable inputs (the empty clause is generated) but may diverge on satisfiable ones. Therefore, the superposition calculus provides a decision procedure for the theory of interest if one can show that it terminates on every input made of the (finitely many) axioms and any set of ground literals. The needed termination proof can be done by hand, by analyzing

*elena.tushkanova@femto-st.fr
†alain.giorgetti@femto-st.fr
‡Christophe.Ringeissen@loria.fr
§olga.kouchnarenko@femto-st.fr

the (finitely many) forms of clauses generated by saturation, but the process is tedious and error-prone. To simplify this process, a schematic superposition calculus has been developed [3] to build the schematic form of the saturations. This schematic superposition calculus is very useful to analyze the behavior of the superposition calculus on a given input theory, as shown in [4] to prove automatically the termination and the combinability of the related decision procedure.

In [5] a schematic superposition calculus for *unit* theories has been considered. The calculus being defined by an inference system we have proposed a rule-based system to execute it. For the implementation the Maude system has been used because it includes support for unification and narrowing, which are key operations of the calculus of interest, and the Maude meta-level provides a flexible way to control the application of rules and powerful search mechanisms.

In this paper we go further and consider the general schematic superposition calculus for *arbitrary* theories, not only its restriction to unit ones. The main contribution is the presentation of a rule-based framework and a tool implementing a complete many-sorted schematic superposition calculus for non-unit theories. The tool allows us to automatically check whether the superposition calculus terminates for theories defined by arbitrary clauses. Moreover, the tool can be used to check modular termination when a combination of signature-disjoint theories is considered.

The design of a schematic paramodulation calculus for general clauses is much more involved than for unit clauses. Indeed the first version proposed in [3] contained a flaw, namely a non-termination issue. This issue was addressed by [4], by considering a new deletion rule specific to non-unit clauses. But that rule did not take into account the constants in the theory signature (such as the constant nil in the signature of the theory of possibly empty lists). One of our contributions is to propose and implement a new schematic paramodulation calculus properly taking these constants into account. Another contribution consists in showing that the schematic superposition calculus halts for the theories in [6], such as the theory of lists (with and without extensionality), the theory of records, the theory of possibly empty lists and the theory of arrays. Moreover, our implementation of schematic superposition provides a trace of each applied rule which is very useful to validate or invalidate saturation proofs previously described in the literature.

Our tool is also very useful for further investigations related to paramodulation calculi extending the standard paramodulation. We present here the case of a (schematic) paramodulation calculus modulo a simple fragment of arithmetic, called Integer Offsets. This extension allows us to consider classical data structures equipped with counting operators. Our tool produces automatically the schematic saturations of the corresponding theories.

The paper is structured as follows. After introducing preliminary notions and presenting paramodulation calculus in Section 2, Section 3 presents a schematic paramodulation calculus that can be used for proving termination of any fair paramodulation strategies and for checking whether paramodulation calculus decides some unions of finitely presented theories. Section 5 explains how we implement the schematic paramodulation calculus using the Maude system. Then Section 6 reports on our experimentations with the tool, to prove the termination of superposition for theories corresponding to classical data structures such as lists, records and arrays. Section 4 reports on our experiments with the extension to Integer Offsets. Finally, Section 7 concludes and presents future work.

2 Background

2.1 First-Order Logic

We consider many-sorted first-order logic with equality. A (many-sorted) *signature* Σ consists of a set of sorts S together with a set of function symbols and a set of predicate symbols. A function symbol is declared using a form $f : s_1 \times \ldots \times s_n \to s$, where $n \geq 0$ is its arity, $s_1, \ldots, s_n$ and s are *sorts* in S. The sorts $s_1, \ldots, s_n$ are called the *argument sorts* and s is called the *value sort* of f. The set of predicate symbols contains only equality symbols of the form $=_s$: $s \times s$ for each $s \in S$. The equality $=_s$ is simply denoted by $=$ when the sort s is clear from the context. Usually, we will just give the signature without explicitly mentioning the equality symbols. For instance, $\Sigma_L = \{\mathsf{nil} : \textsc{lists}, \mathsf{car} : \textsc{lists} \to \textsc{elem}, \mathsf{cdr} : \textsc{lists} \to \textsc{lists}, \mathsf{cons} : \textsc{elem} \times \textsc{lists} \to \textsc{lists}\}$ is a usual signature for lists.

Given a signature Σ, we assume the usual first-order notions of term, position, literal, clause and substitution, as defined, e.g., in [7], and only detail here some notions and notations that are less classical and essential to understand the rest of the paper.

Given a term t and a position p, $t_{|p}$ denotes the subterm of t at position p, and $t[l]_p$ denotes a term t such that $t_{|p} = l$. By a standard abuse of notation, we (ambiguously) denote by $t[r]_p$ the term obtained from t by replacing the subterm at position p by r. When the position p is clear from the context, we may simply write $t[l]$. Application of a substitution σ to a term t is written $\sigma(t)$. The notations $C[l]$ and $\sigma(C)$ are also used for any clause C.

Given a function symbol f, an *f-rooted* term is a term whose top-symbol is f. A *compound* term is an f-rooted term for a function symbol f of positive arity. The *depth* of a term is defined inductively as follows: $\mathit{depth}(t) = 0$, if t is a constant or a variable, and $\mathit{depth}(f(t_1, \ldots, t_n)) = 1 + \mathit{max}\{\mathit{depth}(t_i) \mid 1 \leq i \leq n\}$. A term is *flat* if its depth is 0 or 1. A *positive literal* is an equality $l = r$ and a *negative literal* is a disequality $l \neq r$, where l and r have the same sort. We use the symbol $\bowtie$ to denote either $=$ or $\neq$. The depth of a literal $l \bowtie r$ is defined as follows: $\mathit{depth}(l \bowtie r) = \mathit{depth}(l) + \mathit{depth}(r)$. A positive literal is *flat* if its depth is 0 or 1. A negative literal is *flat* if its depth is 0.

A clause is either the empty clause or a finite disjunction of literals $l_1 \vee \ldots \vee l_n$, where l_i is a literal for each $1 \leq i \leq n$. For instance the disjunction of literals $(c_1 \neq c_2 \vee \mathsf{car}(v) = e)$ is a clause for the above-mentioned signature Σ_L of lists, if c_1 and c_2 are two constants of the same sort, v is a constant of sort LISTS and e is a constant of sort ELEM. A *unit clause* is a clause composed of exactly one literal. For example, for a constant l of sort LISTS, $l = \mathsf{nil}$ is a unit clause. The *empty* clause, i.e. the clause with no disjunct, corresponding to an unsatisfiable formula, is denoted by $\bot$.

Terms, literals and clauses are *ground* whenever no variable appears in them. Throughout this document, universally quantified variables are represented by capital letters.

We also assume the usual first-order notions of model, satisfiability, validity, logical consequence. A *first-order theory* (over a finite signature) is a set of first-order formulae with no free variables. When T is a finitely axiomatized theory, $Ax(T)$ denotes the set of axioms of T. In this paper we consider first-order theories *with equality*, for which the equality symbol $=$ is always interpreted as the equality relation. A formula is *satisfiable in a theory* T if it is satisfiable in a model of T. The *satisfiability problem* modulo a theory T amounts to establishing whether any given finite conjunction of literals (or equivalently, any given finite set of literals) is T-satisfiable or not.

We consider inference systems using well-founded orderings on terms (resp. literals) that are total on ground terms (resp. literals). An ordering $<$ on terms is a *simplification ordering* [7] if it is

stable ($l < r$ implies $\sigma(l) < \sigma(r)$ for every substitution σ), monotonic ($l < r$ implies $t[l]_p < t[r]_p$ for every term t and position p), and has the subterm property (i.e., it contains the subterm ordering: if l is a strict subterm of r, then $l < r$). Simplification orderings are well-founded for finite signatures. A term t is *maximal* in a multiset S of terms if there is no $u \in S$ such that $t < u$, equivalently $t \not< u$ for every $u \in S$. Hence, $t \not\leq u$ (equivalently, $t \not< u$ and $t \neq u$) means that t is maximal in $\{t, u\}$, where t and u are different terms. In the same way as in [2], an ordering on terms is extended to literals by using its multiset extension on literals viewed as multisets of terms: any positive literal $l = r$ (resp. negative literal $l \neq r$) is viewed as the multiset $\{l, r\}$ (resp. $\{l, l, r, r\}$). A literal ℓ is *maximal* in a set S of literals if there is no literal $\ell' \in S$ such that the corresponding multiset of ℓ is smaller than the corresponding multiset of ℓ'. The multiset extension of $<$ is the smallest ordering $<_{mul}$ on multisets of terms such that $M \cup \{t_1, \ldots, t_n\} <_{mul} M \cup \{s\}$ if $t_i < s$ for all $i \in 1 \ldots n$. By a slight abuse of notation, $<_{mul}$ is often also denoted by $<$.

Superposition $\dfrac{C \vee l[u'] \bowtie r \qquad D \vee u = t}{\sigma(C \vee D \vee l[t] \bowtie r)}$

if $\sigma(u) \not\leq \sigma(t)$, $\sigma(l[u']) \not\leq \sigma(r)$, $l[u'] \bowtie r$ and $u = t$ are selected in their clauses.

Reflection $\dfrac{C \vee u \neq u'}{\sigma(C)}$

if $u \neq u'$ is selected in its clause.

Eq. Factoring $\dfrac{C \vee u = t \vee u' = t'}{\sigma(C \vee t \neq t' \vee u = t)}$

if $\sigma(u) \not\leq \sigma(t)$, $u = t$ is selected in its clause, $\sigma(t) \not\leq \sigma(t')$ and $\sigma(u') \not\leq \sigma(t')$.

Above, σ is the most general unifier of u and u' and C and D are clauses.
In the *Superposition* rule, u' is not a variable.

Figure 1: Expansion inference rules of $\mathcal{PC}$

2.2 Paramodulation Calculus

Our presentation of the *Paramodulation calculus* $\mathcal{PC}$ takes the best (to our sense) from the presentations in [1], [3] and [4]. The inference system $\mathcal{PC}$ consists of the rules in Figs. 1 and 2. Expansion rules (Fig. 1) aim at generating new (deduced) clauses. For brevity left and right paramodulation rules are grouped into a single rule, called *Superposition*, that uses an equality to perform a replacement of equal by equal into a literal. The *Reflection* and *Eq. Factoring* rules generate a new clause from the instantiation of an existing one. *Reflection* removes a selected disequality in a clause when the instantiation unifies its two sides. When the clause is unit, it generates the empty clause. *Eq. Factoring* factorizes two equalities when the instantiation unifies their left-hand sides. Contraction rules (Fig. 2) aim at simplifying the set of clauses. Using *Subsumption*, a clause is removed when it is an instance of another one. *Simplification* rewrites a literal into a simpler one by using an equality that can be considered as a rewrite rule. Trivial equalities are removed

Subsumption	$\frac{S \cup \{C, C'\}}{S \cup \{C\}}$
	if for some substitution σ, $\sigma(C) \subseteq C'$.
Simplification	$\frac{S \cup \{C[l'], l = r\}}{S \cup \{C[\sigma(r)], l = r\}}$
	if $l' = \sigma(l)$, $\sigma(l) > \sigma(r)$, and $C[l'] > (\sigma(l) = \sigma(r))$
Tautology	$\frac{S \cup \{C \vee t = t\}}{S}$

Above, C and C' are clauses and S is a set of clauses.

Figure 2: Contraction inference rules of $\mathcal{PC}$

by *Tautology*. Here, unification, pattern-matching and substitution application are many-sorted, i.e. they respect sorts. A fundamental feature of $\mathcal{PC}$ is the usage of a simplification ordering $<$ to control the application of *Superposition* and *Simplification* rules by orienting equalities. Hence, the *Superposition* rule is applied by using terms that are maximal in their literals with respect to $<$. This ordering is total on ground terms. We use a lexicographic path ordering [7] such that terms of positive depth are greater than constants. $\mathcal{PC}$ uses a selection function *sel* such that for each clause C, $sel(C)$ contains a negative literal in C or all maximal literals in C w.r.t. $<$. A literal in C is said *selected* in C if it occurs in the range of $sel(C)$.

Let us recall the usual definitions of redundancy, saturation, derivation and fairness. A clause C is *redundant* with respect to a set S of clauses if either $C \in S$ or S can be obtained from $S \cup \{C\}$ by a sequence of applications of contraction rules. An inference is *redundant* with respect to a set of clauses S if its conclusion is redundant with respect to S. A set of clauses S is *saturated* if every inference with a premise in S is redundant with respect to S. A *derivation* is a sequence $S_0, S_1, \ldots, S_i, \ldots$ of sets of clauses where each S_{i+1} is obtained from S_i by applying an inference to add a clause (by expansion rules in Fig. 1) or to delete a clause (by contraction rules in Fig. 2). For the *Simplification* rule, one can remark that its application corresponds to two steps in the derivation: the first step adds a new literal, whilst the second one deletes a literal. A derivation is characterized by its *limit*, defined as the set of *persistent* clauses $\bigcup_{j \geq 0} \bigcap_{i > j} S_i$, that is, the union for each $j \geq 0$ of the set of clauses occurring in all future steps starting from S_j. A derivation $S_0, S_1, ..., S_i, ...$ is *fair* if for every inference with premises in the limit, there is some $j \geq 0$ such that the inference is redundant with respect to S_j. The set of persistent literals obtained by a fair derivation is called the *saturation* of the derivation.

3 Schematic Paramodulation Calculus

The *Schematic Paramodulation Calculus* aims at computing an abstract form of all saturations generated by the paramodulation calculus. Hence, the termination of the schematic paramodulation calculus for a *single* schematic input implies the termination of the paramodulation calculus for *all*

possible inputs. More generally, the schematic paramodulation calculus is an automated tool to check properties of paramodulation related to termination and combinability [4]. In [5], we have described a first implementation of the schematic paramodulation calculus for the restricted case of unit clauses. Now, we consider the general case of arbitrary clauses for which the design of a schematic paramodulation calculus is much more involved than for unit clauses. Indeed the first version proposed in [3] contained a flaw, namely a non-termination issue. This issue was addressed by [4], by considering a new deletion rule specific to non-unit clauses. Compared to [4], the new schematic paramodulation calculus we present is a slight adaptation that terminates more often for theories having constants in their signature.

At the first glance, the schematic paramodulation calculus, named $\mathcal{SPC}$, is almost identical to $\mathcal{PC}$, except that clauses are constrained. In the following, we first define some useful notions related to this constraint-based framework. Then, $\mathcal{SPC}$ is defined, and its properties are investigated.

3.1 Definitions

This section introduces the notions of constraint, constrained clause, constrained variable and elementary instance. It also gives the intuition behind these notions.

Definition 1 *An* atomic constraint *is of the form $const(t)$, where t is a term. A* constraint *is a conjunction of atomic constraints. A* constrained clause *is of the form $C\|\varphi$, where C is a clause and φ is a constraint. A variable x is* constrained *in a constrained clause $C\|\varphi$ if φ contains $const(x)$; otherwise it is* unconstrained.

In fact, a constrained variable is a schematization of constants. It can only be instantiated by a constant of the same sort, the sort of a variable in a clause being determined by its position in this clause. For instance, the constrained variable x in the constrained clause $\mathsf{car}(x) = e \| const(x) \wedge const(e)$ can only be replaced with a constant of sort LISTS, which is the argument sort of car. Similarly, the constrained variable e in the same clause can only be replaced with a constant of the value sort ELEM of car. An unconstrained variable is a universally quantified variable that can be instantiated by any term of the same sort. For sake of brevity, $const(x_1, \ldots, x_n)$ denotes the conjunction $const(x_1) \wedge \cdots \wedge const(x_n)$.

Definition 2 (Constraint satisfaction) *The atomic constraint $const(t)$ is* true *iff t is a constant. A constraint is* true *if all its atomic constraints are true. A substitution σ* satisfies *a constraint φ if $\sigma(\varphi)$ is true. A constraint is* satisfiable *if there exists a substitution σ satisfying it.*

Consequently, the atomic constraint $const(t)$ is unsatisfiable if t is a term of depth 1 or more, i.e., when t contains a non-constant function symbol. When a constraint contains a true atomic constraint, we assume that this true atomic constraint is automatically removed from the constraint.

Definition 3 (Constraint instance) *A* constraint instance *of $C\|\varphi$ is any clause of the form $\sigma(C)$ where σ is a substitution satisfying φ.*

For example, if a is a constant then the clause $f(a) = X$ is a constraint instance of the constrained clause $f(x) = X \| const(x)$, where x is a constrained variable and X is an unconstrained variable. A constrained clause is used to schematize the set of all its constraint instances.

The notion of constraint instance is extended to constrained clauses by the following definition.

Definition 4 (Elementary instance) *Let $C\|\varphi$ and $C'\|\varphi'$ be two constrained clauses. We say that $C'\|\varphi'$ is an* elementary instance *of $C\|\varphi$ if there exists a substitution σ replacing some constrained variables of $C\|\varphi$ with constrained variables or constants of $C'\|\varphi'$ such that $C' = \sigma(C)$ and $\varphi' = \sigma(\varphi)$.*

For instance, the clause $f(x) = \mathsf{nil} \vee y = g(z) \| const(x, y, z)$ is an elementary instance of the clause $f(a) = b \vee c = g(d) \| const(a, b, c, d)$, because the substitution $\sigma = \{a \leftarrow x, b \leftarrow \mathsf{nil}, c \leftarrow y, d \leftarrow z\}$ satisfies all the conditions in Definition 4.

Notice that our notion of constraint is less general and thus more precise than the one in [4], where an atomic constraint is some $t \leq t'$. Our atomic constraint $const(t)$ corresponds to $t \leq c^T$ in [4], where c^T represents the biggest constant with respect to $\leq$. Since the ordering $\leq$ is extended to open terms, there may exist unground substitutions σ such that $\sigma(t \leq t')$ is true. By contrast, our constraints are satisfiable iff they are satisfiable by a ground substitution that replaces all their variables with constants.

3.2 Schematic Calculus

Superposition
$$\frac{C \vee l[u'] \bowtie r \| \varphi \qquad D \vee u = t \| \psi}{\sigma(C \vee D \vee l[t] \bowtie r \| \varphi \wedge \psi)}$$
if $\sigma(u) \not\leq \sigma(t)$, $\sigma(l[u']) \not\leq \sigma(r)$, $l[u'] \bowtie r$ and $u = t$ are selected in their clauses.

Reflection
$$\frac{C \vee u \neq u' \| \varphi}{\sigma(C \| \varphi)}$$
if $u \neq u'$ is selected in its clause

Eq. Factoring
$$\frac{C \vee u = t \vee u' = t' \| \varphi}{\sigma(C \vee t \neq t' \vee u = t \| \varphi)}$$
if $\sigma(u) \not\leq \sigma(t)$, $u = t$ is selected in its clause, $\sigma(t) \not\leq \sigma(t')$ and $\sigma(u') \not\leq \sigma(t')$.

Above, σ is the most general unifier of u and u' and C and D are clauses. In the *Superposition* rule, u' is not an unconstrained variable.

Figure 3: Expansion inference rules of $\mathcal{SPC}$

$\mathcal{SPC}$ consists of the rules in Figs. 3 and 4. With respect to [4], we have slightly adapted the subsumption rule so that the instantiation is not only a renaming but can also be a substitution instantiating constrained variables by constrained variables or constants. For example, the constrained clause $x \neq \mathsf{nil} \parallel const(x)$ where nil is a constant is subsumed by the constrained clause $a \neq b \parallel const(a, b)$ where a and b are of sort LISTS. This allows us to have a more compact form of saturations.

Subsumption $\dfrac{S \cup \{C\|\varphi, C'\|\varphi'\}}{S \cup \{C\|\varphi\}}$

if a) $C \in Ax(T)$, φ is empty and for some substitution σ, $C' = \sigma(C)$; or b) $C' = \sigma(C)$ and $\varphi' = \sigma(\varphi)$, where σ is a renaming or a mapping from constrained variables to constrained variables

Simplification $\dfrac{S \cup \{C[l']\|\varphi, l = r\}}{S \cup \{C[\sigma(r)]\|\varphi, l = r\}}$

if i) $l = r \in Ax(T)$, ii) $l' = \sigma(l)$, iii) $\sigma(l) > \sigma(r)$, and $C[l'] > (\sigma(l) = \sigma(r))$

Tautology $\dfrac{S \cup \{C \vee t = t\|\varphi\}}{S}$

Deletion $\dfrac{S \cup \{C\|\varphi\}}{S}$ **if** φ is unsatisfiable

Above, $C\|\varphi$ and $C'\|\varphi'$ are constrained clauses and S is a set of constrained clauses.

Figure 4: Contraction inference rules of $\mathcal{SPC}$

For a given theory whose signature is Σ and set of sorts is S, let

$$\begin{array}{rl} G_0 = & \{\bot\} \cup \bigcup_{s \in S} \{x =_s y \parallel const(x, y), x \neq_s y \parallel const(x, y)\} \\ & \cup \bigcup_{f \in \Sigma} \{f(x_1, \ldots, x_n) = x_0 \parallel const(x_0, x_1, \ldots, x_n)\} \end{array}$$

where $n \geq 1$. This set schematizes any set of ground flat equalities and disequalities built over Σ, along with the empty clause. The procedure for checking termination of any fair paramodulation strategy is based on *Schematic Saturation*, which consists in executing $\mathcal{SPC}$ on $Ax(T) \cup G_0$. If $\mathcal{SPC}$ halts on $Ax(T) \cup G_0$, then $\mathcal{PC}$ halts on $Ax(T) \cup S$, for any arbitrary set S of ground flat literals. This property will be proved in Sect. 3.4. A key ingredient is the Schematic Deletion rule defined in the next section.

3.3 Schematic Deletion Rule

The schematic saturation may generate longer and longer clauses (containing new constrained variables) from clauses containing unconstrained variables and therefore diverge. To illustrate this fact let us consider two examples, namely the theory of arrays and the theory of possibly empty lists. As in [6], we consider here mono-sorted versions of these two theories.

The theory of arrays is axiomatized by the following set $Ax(A)$ of axioms:

$$\mathsf{select}(\mathsf{store}(A, I, E), I) = E \quad (1)$$
$$\mathsf{select}(\mathsf{store}(A, I, E), J) = \mathsf{select}(A, J) \vee I = J \quad (2)$$

For every set S of ground flat literals, any saturation of $Ax(A) \cup S$ is finite [6], while schematic saturation diverges [4]. In fact, superposition between the axiom (2) and $\mathsf{store}(a_1, i, e) = a_2 \parallel const(a_1, i, e, a_2)$ generates the clause $\mathsf{select}(a_2, J) = \mathsf{select}(a_1, J) \vee i = J \parallel const(a_1, a_2, i)$ whose superposition with a renamed copy of itself generates a clause of a new form $\mathsf{select}(x, J') = \mathsf{select}(a_1, J') \vee i = J' \vee z = J' \parallel const(x, a_1, i, z)$. This process continues to generate longer and longer clauses so that the schematic saturation does not terminate.

The theory of possibly empty lists (*PEL* for short) is axiomatized by the following set $Ax(PEL)$ of axioms:

$$\mathsf{car}(\mathsf{cons}(X, Y)) = X \qquad \mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)) = Y \vee Y = \mathsf{nil}$$
$$\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y \qquad \mathsf{car}(\mathsf{nil}) = \mathsf{nil}$$
$$\mathsf{cons}(X, Y) \neq \mathsf{nil} \qquad \mathsf{cdr}(\mathsf{nil}) = \mathsf{nil}$$

The schematic saturation generates the clause $\mathsf{cons}(x, \mathsf{cdr}(y)) = z \vee z = \mathsf{nil}$, whose superposition with a renamed copy of itself generates a clause $z = z' \vee z = \mathsf{nil} \vee z' = \mathsf{nil}$. This process goes on to generate longer and longer clauses so that the schematic saturation diverges as well.

A *Schematic Deletion* rule has been designed [4] to cope with this problem. We adapt this rule to take into account the constants in the theory signature, such as the constant nil for the theory of possibly empty lists. The new version of the *Schematic Deletion* rule is composed of the two rules in Fig. 5. The idea behind these rules is to delete 1) disjunctions of two or more equalities and disequalities between two constrained variables or between a constrained variable and a constant, and 2) constrained clauses composed of an elementary instance $D \vee l \| \varphi$ of some other constrained clause and literals l_i which are not maximal in $D \vee l$ and are elementary instances of $l \| \varphi$. In case 1) the clause is deleted because its (dis)equalities may superpose with themselves to generate infinitely many disjunctions of (dis)equalities between constrained variables or between a constrained variable and a constant. In case 2) the clause is deleted because superposition between this clause and itself may generate infinitely many new clauses of the same kind.

3.4 Adequation Result

Let us now present the result stating that every clause in a saturation corresponds to a schematic clause in a schematic saturation. This result was initially proved for the schematic calculus considered in [4]. The same result holds for the schematic calculus $\mathcal{SPC}$ considered here.

Theorem 1 (Correspondence between $\mathcal{PC}$ and $\mathcal{SPC}$) *Let T be a theory axiomatized by a finite set $Ax(T)$ of clauses, which is saturated with respect to $\mathcal{PC}$. Let G_∞^T be the set of all clauses in a saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$. Then for every set S of ground flat Σ_T-literals, every clause in a saturation $Ax(T) \cup S$ by $\mathcal{PC}$ is a clause of the form*

$$C \vee l_1 \vee \ldots \vee l_n \quad (*)$$

where

- $n \geq 0$, *and*

Sch. Deletion1 $\dfrac{S \cup \{C\|\varphi\}}{S}$

if $C\|\varphi$ is a non-unit clause containing only equalities or disequalities between constrained variables or between a constrained variable and a constant.

Sch. Deletion2 $\dfrac{S \cup \{D'\|\varphi'\} \cup \{D \vee l \vee l_1 \vee \ldots \vee l_n\|\varphi\}}{S \cup \{D'\|\varphi'\}}$

if $n \geq 0$, $D \vee l\|\varphi$ is an elementary instance of the clause $D'\|\varphi'$, and $l_i\|\varphi$ is an elementary instance of $l\|\varphi$, where l is a non-maximal literal in $D \vee l$, for $i = 1, \ldots, n$

Figure 5: Schematic Deletion rules of $\mathcal{SPC}$

- *C is a constraint instance of some clause C' in G^T_∞, and*
- *l_i is*
  - *either a constraint instance of some non-maximal literal in C', or else*
  - *a constraint instance of some maximal (dis)equality between constrained variables in C', or else*
  - *a non-maximal (dis)equality between constants.*

Proof. The proof in [4] can be replayed in the same way. The proof is by induction on the length of derivations of $\mathcal{PC}$. The base case is obvious. For the inductive case, we need to show all the three facts:

1. Each clause added in the process of saturation of $Ax(T) \cup S$ by $\mathcal{PC}$ is of the form (*). This is true because we use the same expansion rules as in [4].

2. If a constrained clause $C\|\varphi$ is deleted by *Subsumption* or by *Tautology Deletion* from (or simplified by *Simplification* in) the saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$, then all clauses containing a constraint instance of $C\|\varphi$ will also be deleted from (or simplified in) the saturation of $Ax(T) \cup S$ by $\mathcal{PC}$. Compared to [4], only *Subsumption* has slightly changed, and this new contraction rule still satisfies this fact.

3. If a constrained clause $C\|\varphi$ is deleted by *Schematic Deletion* from the saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$, then all constraint instances of this constrained clause $C\|\varphi$ are of the form (*). Compared to [4], we have now to consider the two cases of our new *Schematic Deletion* rule. In the first case, the fact that $l_1 \vee \ldots \vee l_n\|\varphi$ is a non-unit clause containing only equalities or disequalities between terms of depth 0 means that it is a schematization of disjunctions of (dis)equalities between constants. It is easy to see that any disjunction of (dis)equalities between constants is of the form (*). In the second case, the fact that $D \vee l\|\varphi$ is an elementary instance of some clause $D'\|\varphi'$, and $l_i\|\varphi$ is an elementary instance of some non-maximal literal $l\|\varphi$ in $D \vee l$ means that any constraint instance of $D \vee l \vee l_1 \vee \ldots \vee l_n\|\varphi$ is of the form (*). □

3.5 Automatic Combinability

In [8] a rewriting-based approach is used to combine signature-disjoint theories. This approach addresses an interesting modularity problem: if $\mathcal{PC}$ halts for both theory T_1 and theory T_2, can one conclude that $\mathcal{PC}$ halts for $T_1 \cup T_2$? In this case, the only problem that can prevent the termination of $\mathcal{PC}$ for the union of the two theories comes from inferences across theories, since a variable can superpose with any non-variable subterm. To circumvent this problem and ensure modular termination, it is sufficient to exclude inferences on variables across theories. To identify the clauses generating these undesirable inferences, the concept of variable-active clause has been introduced in [8]. In [4] this notion has been extended to a constrained clause.

Definition 5 (Variable-active Clause, Combinable Theory) *A clause C is* variable-active *with respect to an ordering $<$ if C contains a maximal (with respect to $<$) literal of the form $X = t$, where X is a variable not occurring in the set $Var(t)$ of variables of t. A constrained clause is* variable-active *with respect to $<$ if one of its constraint instances is variable-active with respect to $<$. Let T be a theory axiomatized by a finite set $Ax(T)$ of clauses, which is saturated with respect to $\mathcal{PC}$. The theory is said* combinable with $\mathcal{PC}$ *if for every set S of ground flat literals, any saturation of $Ax(T) \cup S$ by $\mathcal{PC}$ is finite and does not contain any variable-active clauses.*

Given two signature-disjoint theories T_1 and T_2, if T_1 and T_2 are combinable with $\mathcal{PC}$, then $\mathcal{PC}$ is a satisfiability procedure for $T_1 \cup T_2$, as shown in [4, Theorem 6]. The correspondence between $\mathcal{SPC}$ and $\mathcal{PC}$ provides us a way to automatically check that a theory T is combinable with $\mathcal{PC}$.

Lemma 1 ([4]) *A theory T is combinable with $\mathcal{PC}$ if any saturation of $Ax(T) \cup G_0$ by $\mathcal{SPC}$ is finite and does not contain any variable-active clauses.*

The tool supporting the proposals of this article provides a function to check that a saturation by $\mathcal{SPC}$ does not contain any variable-active clauses. When this property holds, one can conclude that the considered theory is combinable with $\mathcal{PC}$.

4 Schematic Paramodulation Modulo a Fragment of Arithmetic

Our proof system has been used to perform experiments for an extension of the classical paramodulation calculus. The considered extension allows us to have built-in axioms in the calculus, and so to design paramodulation calculi modulo theories. This is particularly important for arithmetic fragments due to the ubiquity of arithmetics in applications of formal methods. For instance, paramodulation calculi have been developed for Abelian Groups [9, 10] and Integer Offsets [11]. In [11], the termination of paramodulation modulo Integer Offsets is proved manually. Therefore, there is an obvious need for a tool support to automatically prove that an input theory admits a decision procedure based on paramodulation modulo Integer Offsets. We show that our proof system can be adapted to consider this extension to Integer Offsets.

In this section, we introduce theoretical underpinnings that allow us to automatically prove the termination of paramodulation modulo Integer Offsets. To this aim, we design a new schematic paramodulation calculus to describe saturations modulo Integer Offsets. Our approach requires a new form of schematization to cope with arithmetic expressions. As seen in Section 3.4, the interest of schematic paramodulation relies on a correspondence between a derivation using (concrete) paramodulation and a derivation using schematic paramodulation: Roughly speaking, the

set of derivations obtained by schematic paramodulation over-approximates the set of derivations obtained by (concrete) paramodulation. As explained in [12], the fact of considering Integer Offsets requires some specific proof arguments in order to state that the termination of schematic paramodulation implies the termination of (concrete) paramodulation.

4.1 Paramodulation Calculus for Integer Offsets

The paramodulation calculus $\mathcal{UPC}_I$ defined in [11] adapts the paramodulation calculus $\mathcal{PC}$, in the case of unit clauses, to the theory of Integer Offsets, so that it can serve as a basis for the design of decision procedures for data structures equipped with counting operators, such as for instance the theory of lists with length. Technically, the axioms of the theory of Integer Offsets are directly integrated in the simplification rules of $\mathcal{UPC}_I$. The theory of Integer Offsets is axiomatized by the set of axioms $\{\forall X.\ \mathsf{s}(X) \neq 0,\ \forall X, Y.\ \mathsf{s}(X) = \mathsf{s}(Y) \Rightarrow X = Y,\ \forall X.\ X \neq \mathsf{s}^n(X) \ \text{ for all } \ n \geq 1\}$ over the signature $\Sigma_I := \{0 : \text{INT}, \mathsf{s} : \text{INT} \rightarrow \text{INT}\}$. A possible Integer Offsets extension is the theory LLI of lists with length whose signature is $\Sigma_{LLI} = \{\mathsf{car} : \text{LISTS} \rightarrow \text{ELEM}, \mathsf{cdr} : \text{LISTS} \rightarrow \text{LISTS}, \mathsf{cons} : \text{ELEM} \times \text{LISTS} \rightarrow \text{LISTS}, \mathsf{len} : \text{LISTS} \rightarrow \text{INT}, \mathsf{nil} :\rightarrow \text{LISTS}, 0 :\rightarrow \text{INT}, \mathsf{s} : \text{INT} \rightarrow \text{INT}\}$ and whose set of axioms $Ax(LLI)$ is $\{\mathsf{car}(\mathsf{cons}(X, Y)) = X, \mathsf{cdr}(\mathsf{cons}(X, Y)) = Y, \mathsf{cons}(X, Y) \neq \mathsf{nil}, \mathsf{len}(\mathsf{cons}(X, Y)) = \mathsf{s}(\mathsf{len}(Y)), \mathsf{len}(\mathsf{nil}) = 0\}$.

4.2 Schematic Paramodulation Calculus for Integer Offsets

This section briefly introduces a new schematic calculus $\mathcal{SUPC}_I$. It is a schematization of $\mathcal{UPC}_I$ taking into account the axioms of the theory of Integer Offsets within the framework based on schematic paramodulation.

The theory of Integer Offsets allows us to build arithmetic expressions of the form $\mathsf{s}^n(t)$ for $n > 0$. The idea investigated here is to represent all terms of this form in a unique way. To this end, we consider a new operator $\mathsf{s}^+ : \text{INT} \rightarrow \text{INT}$ such that $\mathsf{s}^+(t)$ denotes the infinite set of terms $\{s^n(t) \ \mid \ n > 0\}$. In this new context, a *schematic clause* is a constrained clause built over the signature extended with s^+, and an *instance of a schematic clause* is a constraint instance where each occurrence of s^+ is replaced by some s^n with $n > 0$. The calculus $\mathcal{SUPC}_I$ takes as input a set of schematic literals, G_0^+, that extends G_0 with a new form of literals:

$$G_0^+ = \quad G_0 \cup \{u = \mathsf{s}^+(v) \| const(x_0, x_1, \ldots, x_n)\}$$

where u, v are flat terms of sort INT whose variables $x_0, \ldots, x_n$ are all constrained.

The calculus $\mathcal{SUPC}_I$ re-uses most of the rules of $\mathcal{SPC}$, restricted to the case of unit clauses, and complete them with some additional reduction rules. It is important to note that $\mathcal{SUPC}_I$ uses a specific term rewrite system to simplify schematic terms, and a specific schematic deletion rule to avoid divergence. Let us detail these two ingredients of $\mathcal{SUPC}_I$.

Whenever a literal is generated by superposition or simplification, the rewrite system $R\mathsf{s}^+ = \{\ \mathsf{s}^+(\mathsf{s}(x)) \rightarrow \mathsf{s}^+(x),\ \mathsf{s}(\mathsf{s}^+(x)) \rightarrow \mathsf{s}^+(x),\ \mathsf{s}^+(\mathsf{s}^+(x)) \rightarrow \mathsf{s}^+(x)\ \}$ is applied eagerly to simplify terms containing s^+. For each of these rules, one can easily check that the set of terms denoted by the left-hand side is included in the set of terms denoted by the right-hand side.

To illustrate the need of a new schematic deletion rule, let us take a look at the theory of lists with length. In that case, it is possible to generate a schematic clause $\mathsf{len}(a) = \mathsf{s}(\mathsf{len}(b)) \| const(a, b)$ which will superpose with a renamed copy of itself, i.e. with $\mathsf{len}(a') = \mathsf{s}(\mathsf{len}(b'))\ \| const(a', b')$ to generate a schematic clause of a new form $\mathsf{len}(a) = \mathsf{s}(\mathsf{s}(\mathsf{len}(b')))\ \| const(a, b')$. This process continues

to generate deeper and deeper schematic clauses so that the calculus will diverge. To avoid this problem, we consider an additional schematic deletion rule to check whether a schematic literal, say $\mathsf{len}(a) = \mathsf{s}^n(\mathsf{len}(b'))\|const(a, b')$, is an instance of an existing schematic literal. Fortunately, G_0^+ already contains $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{len}(b))\|const(a, b)$ which subsumes $\mathsf{len}(a) = \mathsf{s}^n(\mathsf{len}(b'))\|const(a, b')$, and so the latter is deleted.

Similarly to Section 3.4, we are interested in satisfying the following properties:

- Any clause in a saturation generated by $\mathcal{UPC}_I$ with any possible input is an instance of a schematic clause in a saturation generated by $\mathcal{SUPC}_I$ with the input G_0^+.
- The termination of $\mathcal{SUPC}_I$ with the input G_0^+ implies the termination of $\mathcal{UPC}_I$ with any possible input.

The new form of schematization introduced for arithmetic expressions requires adapting the proofs done for the standard case. These new proofs are not detailed in this short introduction to $\mathcal{SUPC}_I$ but can be found in [12]. As shown in [12], we need additional assumptions that are satisfied for theories experimented in Section 6.4.

5 Implementation

This section briefly introduces the Maude language used to implement our schematic paramodulation calculus $\mathcal{SPC}$. Then it describes the main ideas and principles of this implementation.

The tool takes as input a theory and an initial set of constrained clauses. The user has to declare the sorts and the signature of the theory, its set of axioms Ax, and the set G_0 of initial constrained clauses (in separate Maude files with appropriate names). Then a single Maude command saturates the union of Ax and G_0 and displays the saturated set in a pretty way.

With three thousands lines of code instead of one thousand, the tool significantly extends the one for the unit case presented in [5]. For the general case, more rules are implemented, notably the new rule of schematic deletion. The ordering on terms is extended to literals and clauses, and adapted to constrained variables. The tool additionally supports sorts and provides traces of rule applications. It is freely accessible online [13].

5.1 Maude

Maude [14] is a rule-based language well-suited to implement inference systems. Maude's basic programming statements are equations and rules. Its semantics is based on rewriting logic where terms are reduced by applying rewrite rules. Maude has many important features such as reflection, pattern-matching, unification and narrowing. Reflection is a very desirable property of a computational system, because a reflective system can access its own meta-level and this way can be much more powerful, flexible and adaptable than a non-reflective one. Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective. Narrowing [15] is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces pattern-matching by unification in order to (non-deterministically) instantiate and reduce a term. The narrowing feature is provided in an extension of Maude named Full Maude. It is clearly of great interest to implement the superposition rules of our calculus.

5.2 Tool architecture

The tool takes as input a piece of Maude code, hereafter called the input file. The input file declares a set of sorts, a functional signature Σ, a precedence ordering over symbols in Σ, a set of axioms and the initial set of constrained clauses (G_0 or G_0^+) corresponding to the signature, as defined in Sections 3 and 4. The axioms are declared globally by a symbol named `ax`. The precise syntax of input files is easy to infer from examples and templates provided on the tool website [13]. A minimal knowledge of Maude syntax is sufficient to write these files.

The tool itself consists of two parts. The first part is loaded by Maude before the input file. It contains all the definitions needed to support the syntax of input files. In particular, it defines the Maude sorts `Literal`, `SetLit`, `Clause`, `Constraint` and `SClause` to represent respectively literals, set of literals, clauses, constraints and constrained clauses. Its also defines traces as detailed in Section 5.3. The second part is loaded after the input file. It contains the schematic superposition calculus itself. Its interpretation by Maude computes (whenever possible) the saturation of the set composed of the axioms and the initial set of constrained clauses described in the input file. Then, a simple Maude command can check combinability on the saturated set, which is the tool output. More precisely, there are two versions of the second part, respectively implementing the schematic superposition calculi $\mathcal{SPC}$ and $\mathcal{SUPC}_I$. The second one is for theories extending the theory of Integer Offsets. It is selected when the input file contains the sort `Ints`.

5.3 Traces

An important feature of our tool consists in providing a trace indicating the name of the applied rule and the constrained clauses it is applied to at each derivation step. This trace helps understanding the origin of each new constrained clause. With this information, the user could replay the derivation manually if necessary.

Each constrained clause carries its trace. For example the expression

$$\texttt{sup(}C_1\texttt{, }C_2\texttt{, }u\texttt{, }l[u']\texttt{, }Ctx\texttt{) gives }C_3$$

means that the constrained clause $C_3 = \sigma(C \vee D \vee l[t] \bowtie r \| \varphi \wedge \psi)$ is derived from the constrained clauses $C_1 = (C \vee l[u'] \bowtie r \| \varphi)$ and $C_2 = (D \vee u = t \| \psi)$ by superposing the term u from C_2 in the term $l[u']$ from C_1 at the context $Ctx = l[\,]$, where the rewriting has taken place.

The sorts `STrace`[1] and `TracedSClause` of traces and traced constrained clauses are defined by

```
sort STrace .
sort TracedSClause .
subsort SClause < STrace .

op sup : STrace STrace Substitution Substitution -> STrace .
op sup : STrace STrace Term Term Context -> STrace .
op refl : STrace -> STrace .
op ef : STrace -> STrace .
op simpl : STrace STrace -> STrace .

op _gives_ : STrace SClause -> TracedSClause .
```

[1] A letter `S` is added to the Maude sort name for traces because a Maude sort named `Trace` already exists.

The subsort condition considers constrained clauses as traces. Thanks to this condition, the initial constrained clauses and axioms are considered as traced by themselves. The operators `sup`, `refl` and `ef` associate a trace with each expansion rule (respectively, superposition, reflection and equality factoring). One can remark that the operator `sup` has two profiles. The first one is an internal intermediate format introduced for technical reasons. Only the second one is used when outputting traces. It shows not only the traces of the superposed constrained clauses but also the terms to which superposition has been applied, and the context of the application.

The operator `simpl` associates a trace with the *Simplification* rule, a contraction rule that does not eliminate a clause but rewrites it into a simpler one. The infix operator `gives` builds a traced constrained clause from a trace and a constrained clause.

5.4 Inference Rules

This section presents the encoding of $\mathcal{SPC}$. The encoding of $\mathcal{SUPC}_I$ is similar, and therefore omitted. Let us emphasize two main ideas of this encoding: 1) inference rules are translated into rewrite rules, and 2) rule application is controlled thanks to specially designed states. The encoding description starts with the translation of the contraction rules into rewrite rules. Then, it continues with the expansion rules, whose fair application strategy is encoded by using a notion of state together with rules to specify the transitions between states.

5.4.1 Contraction Rules

We first present the encoding of some of the rules that remove a (redundant) clause. The inference rule *Tautology* is simply encoded by the rewrite rule

```
rl [tautology] : Tr gives clause((SL, U equals U)) || Phi
                 => empty .
```

where `Tr` is a clause trace, `SL` is a set of literals, `U` is a term and `Phi` is a constraint. A constant `empty` denotes an empty set. This rule removes a traced constrained clause if it contains a trivial equality.

The inference rule *Deletion* is encoded by the conditional rewrite rule

```
crl [del] : Tr gives C || Phi => empty
  if isSatisfiableSet(Phi) == false .
```

where the function `isSatisfiableSet` checks if a given constraint holds, i.e. none of the terms it constraints is compound. The first case of *Schematic Deletion* inference rule is encoded by

```
crl [sd1] : Tr gives C || Phi => empty
  if condition1(C || Phi) .
```

It removes clauses containing only equalities and disequalities between terms of depth 0. In such literals all the variables are necessarily constrained. The function `condition1` checks these requirements. The second case of *Schematic Deletion* requires a more sophisticated condition which is not detailed here for sake of conciseness. The first case of the *Subsumption* inference rule uses a global variable `ax` that represents the set of axioms of the current theory:

```
crl [subsum1] : Tr gives C || Phi => empty
  if ax isSubsum (Tr gives C || Phi) .
```

The rule condition checks whether the clause `C` can be subsumed by one of the axioms in `ax`. The other two cases of *Subsumption* rule are encoded similarly and therefore omitted.

Simplification is encoded as another conditional rewrite rule using the set of axioms `ax`. This rule reduces a clause to a simpler one thanks to some equality axiom:

```
var newTC : TracedSClause .

crl [simpl] : Tr gives C || Phi
              => simpl(Tr, axTr) gives newC || newPhi
  if newTC := applyUnitAx(C, Phi, ax) /\ newTC =/= NoSimpl /\
     (axTr gives newC || newPhi) := newTC .
```

The function `applyUnitAx` considers each literal in the clause `C` and tries to rewrite it into a simpler one by using one of the equality axioms from `ax` as a rewrite rule. It returns a simplified constrained clause whose trace describes the axiom. If *Simplification* does not apply the function `applyUnitAx` returns the constant `NoSimpl` of sort `TracedSClause`. The last condition is a matching equation decomposing by pattern-matching the constrained clause `newTC` into a trace `axTr`, a clause `newC` and a constraint `newPhi`.

5.4.2 States for Rule Application Control

The order of rule applications has to be controlled. In particular, contraction rules should be given a higher priority than expansion ones. An expected solution could be to control rule applications with the strategy language described in [16, 17], but unfortunately it appeared not to be compatible with the Full Maude version 2.5b required for narrowing (see details in Sect. 5.4.3). To circumvent this technical problem we propose to control rules with states.

In order to detect redundant clauses generated by expansion rules, we consider two distinct states defined as follows:

```
sort State .
op state : SetTracedSClause -> State .
op _redundancy_ : SetTracedSClause TracedSClause -> State .
```

The input state of the expansion rules of $\mathcal{SPC}$ is expected to be of the form `state(`S`)` where S is a set of traced constrained clauses. A state of the form `_redundancy_` is entered after each application of an expansion rule. The state S `redundancy` C is the input state for checking whether the constrained clause C is redundant with respect to the set of constrained clauses S. If this is the case, the clause is added to the set and this leads to a new state of the form `state(`$S \cup C$`)`. Otherwise, the next state is `state(`S`)`.

5.4.3 Superposition Rule

The *Superposition* rule produces a new clause of the form $\sigma((C \vee D \vee l[t] \bowtie r) \parallel \varphi \wedge \psi)$ from any set containing two constrained clauses of the form $(C \vee l[u'] \bowtie r) \| \varphi$ and $(D \vee u = t) \| \psi$, if the side conditions given in Fig. 3 are satisfied with the most general unifier σ of u and u'. This notion of superposition is close to the notion of narrowing. The idea is to use the literal $u = t$ from the second clause as a rewriting rule $u \to t$ to narrow the left-hand side term $l[u']$ of some literal in the first clause. If the narrowing succeeds it produces the term $\sigma(l[t])$. It remains to apply σ to

the right-hand side term r of the literal in the first clause, to the clauses C and D, and to the conjunction of the two constraints φ and ψ.

To narrow we use a function `metaENarrowShowAll` implemented in Full Maude. In the standard version the narrowing is restricted to non-variable positions, along its standard definition. But the *Superposition* rule of $\mathcal{SPC}$ requires an unusual feature: narrowing should also be applied at the positions of the variables schematizing constants. This is why we use a dedicated version of Full Maude provided by Santiago Escobar to implement this unusual feature.

A second difficulty is that the `metaENarrowShowAll` function applied to the term $l[u']$ and the rule $u \to t$ generates all the possible narrowings at all the positions, whereas one application of the *Superposition* rule should produce only one clause. To consider one by one each candidate clause (among the set of results obtained by narrowing) we introduce an additional state `add_withLitFrom_to_` defined by

```
op add_withLitFrom_to_ : TracedSClause SetLSC SetTracedSClause
                         -> State .
```

where the sort `SetLSC` corresponds to the set of results computed by narrowing. The state "`add` C `withLitFrom` N `to` S" stores a part C of the new clause under construction, the set N of narrowing results and the current set of clauses S. Its use is detailed below.

The implementation of the *Superposition* rule is illustrated by Figure 6 showing the intermediate states and guarded transitions. In this figure, the superposition rule applies between two clauses. We start from the state containing an initial set S of clauses. This set is decomposed into $T \cup \{L_1 \vee C_1, L_2 \vee C_2\}$, where L_1 and L_2 are two literals to which superposition will be applied, C_1 and C_2 are two remaining clauses, and T is a set of clauses. After applying narrowing between L_1 and L_2 (transition `[sup2]` in Figure 6), a new state `add` $C_1 \vee C_2$ `withLitFrom` U `to` S is constructed, where $C_1 \vee C_2$ is the disjunction of the remaining clauses, U is the set of all the possible narrowings of L_1 by L_2 at all the positions, and S is the initial set of clauses. If U is empty then by transition `[no-sup]` in Figure 6 we go back to the input state. Otherwise the transition `[select]` in Figure 6 selects some literal L in the set U of narrowing results decomposed into $\{L\} \cup R$, where R is the set of other narrowing results. If the constraint of L obtained from narrowing is satisfiable, then the transition `[select]` (`then` part) builds a new state S `redundancy` N, where N is the new clause $C_1 \vee C_2 \vee L$. Otherwise, another narrowing result is considered (transition `[select]` (`else` part)). If the new clause is redundant with respect to S, i.e. S can be obtained from $S \cup \{C\}$ by a sequence of applications of contraction rules ($S \cup \{C\} \ \to^* \ S$), then the state S remains unchanged (transition `[pick]` (`else` part)). Otherwise, the new clause N is added to the state (transition `[pick]` (`then` part)).

These transitions are implemented by Maude rewriting rules. So, the *Superposition* rule is encoded by five rules named `sup1`, `sup2`, `select`, `no-sup` and `pick`, where `sup1` applies *Superposition* to a clause and itself.

5.4.4 Reflection and Eq. Factoring Rules

The implementation of the *Reflection* rule is divided into two cases. Let us consider the case where the clause consists of only one disequality:

```
crl [reflection1] :
  state((STSC, Tr gives clause(U' != U) || Phi)) =>
  state((STSC, Tr gives clause(U' != U) || Phi,
    emptySClause gives emptySClause))
```

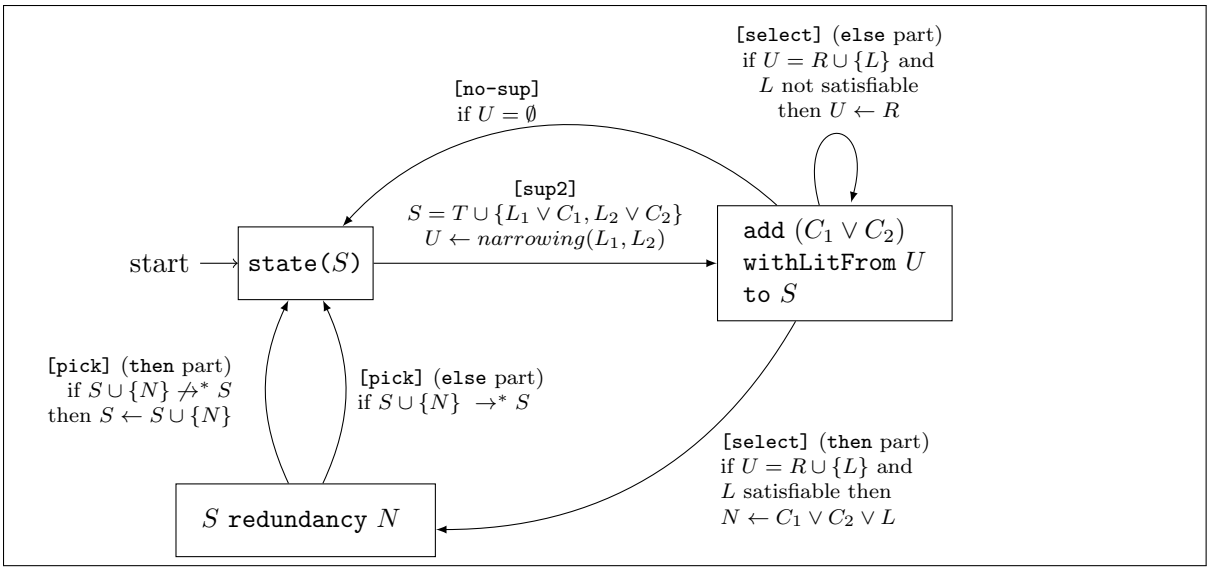

Figure 6: Intermediate states and transitions

```
if isSatisf(U' != U, Phi) .
```

In this case the empty schematic clause `emptySClause` is added to the state (with itself as trace). The function `isSatisf` checks whether two sides of a given disequality are unifiable by a substitution that also satisfies the constraint.

The *Reflection* rule in the general case, when the clause is not a unit one, and the *Eq. Factoring* rule are implemented in a classical way and are therefore not presented here.

5.5 Saturation

A forward search to find generated sets of traced constrained clauses is performed by a function `searchState`. The function call `searchState(St,N)` tries to reach the `N`-th state from an initial state `St` by applying the expansion rules. It performs a breadth-first exploration of the reachable state space.

Then the principle of saturation is implemented by a function `saturate` which implements a fixpoint algorithm in order to reach a state where the set of constrained clauses is saturated. If the initial state is already saturated, then the function returns it unchanged.

A saturated set of constrained clauses could alternatively be computed from an initial state by the Maude `metaSearch` function with a `'!` parameter (searching for a state that cannot be further rewritten), but the function `searchState` computing intermediate states is also interesting for debugging purposes.

5.6 Orderings

A fundamental feature of our superposition calculi is the usage of a simplification ordering which is total on ground terms. This section presents the orderings used in the side conditions of the inference rules and describes their implementation. In our calculus, we assume that compound terms are greater than constants. To satisfy this assumption, it is sufficient to use an LPO ordering

with a precedence on function symbols such that non-constant function symbols are greater than constants.

Definition 6 *Given a precedence $>_F$ on function symbols, the* ***lexicographic path ordering (LPO)*** *$>_{lpo}$ [7] is defined as follows:*

$$\textit{LPO1} \qquad \frac{(s_1,\ldots,s_n) >_{lpo}^{lex} (t_1,\ldots,t_m) \qquad f(s_1,\ldots,s_n) >_{lpo} t_1,\ldots,t_m}{f(s_1,\ldots,s_n) >_{lpo} f(t_1,\ldots,t_m)}$$

$$\textit{LPO2} \qquad \frac{f \;>_F\; g \qquad f(s_1,\ldots,s_n) >_{lpo} t_1,\ldots,t_m}{f(s_1,\ldots,s_n) >_{lpo} g(t_1,\ldots,t_m)}$$

$$\textit{LPO3} \qquad \frac{u_k \;>_{lpo}\; t}{f(u_1,\ldots,u_k,\ldots,u_p) >_{lpo} t}$$

$$\textit{LPO4} \qquad \frac{}{f(u_1,\ldots,u_k,\ldots,u_p) >_{lpo} u_k}$$

where f and g are two functional symbols, $n \geq 0$ and $m \geq 0$ are two non-negative integers, $p \geq 1$ is a positive integer, and s_1, …, s_n, t_1, …, t_m, u_1, …, u_p, t are terms. We write $s >_{lpo} t_1,\ldots,t_m$ when $s >_{lpo} t_k$ for any positive integer $k \in [1,m]$. The ordering $>_{lpo}^{lex}$ denotes the lexicographic extension of $>_{lpo}$.

The LPO ordering is implemented as a Boolean function `gtLPO` such that $\texttt{gtLPO}(s,\texttt{SC},t) = \texttt{true}$ if and only if $s >_{lpo} t$. The additional parameter `SC` collects the constrained variables, that should be viewed as constants in the ordering definition. A dedicated rule defines that compound terms are greater than constrained variables.

For instance, the rule *LPO1* for $n, m \geq 1$ is simply encoded by

```
ceq gtLPO(F[NeSL], SC, F[NeTL]) = true
  if gtLexLPO(NeSL, SC, NeTL) and termGtList(F[NeSL], SC, NeTL) .
```

where `NeSL` and `NeTL` are non-empty lists of terms. In this rule the head symbols of s (= `F[NeSL]`) and t (= `F[NeTL]`) are equal. Then the list of subterms `NeSL` of s should be greater than the list of subterms `NeTL` of t and the term s should be greater than all the elements in the list of subterms of t.

The ordering $>_{lpo}$ on terms is extended to literals thanks to the multiset extension of $>_{lpo}$. An equality $l = r$ is represented as a multiset $\{l, r\}$ while a disequality $l \neq r$ is represented as a multiset $\{l, l, r, r\}$. The lexicographic and multiset extensions of $>_{lpo}$ specified as inference systems are similarly encoded in Maude.

5.7 Automatic Combinability

Checking the combinability of a theory reduces to checking the existence of a variable-active clause in the saturation of G_0 for this theory (see Section 3.5). For the set of maximal literals in a clause, the following function detects whether the variable X does not occur in t when this literal is of the form $X = t$:

```
eq isVarActiveClause(empty, Phi) = false .
eq isVarActiveClause((X equals T, SL), Phi) =
  if not (X inTL vars(T)) and not (X inTL varsOfSC(Phi)) then
    true
  else isVarActiveClause(SL, Phi) fi .
eq isVarActiveClause((L, SL), Phi) =
  isVarActiveClause(SL, Phi) [owise] .
```

6 Experimentation

We have done some experiments to compare the (schematic) saturations computed by our tool with corresponding results in the literature. For unit (mono-sorted) theories such as the theory of lists (with and without extensionality) and the theory of records a comparison can be found in [5]. In Sect. 6.1 we consider many-sorted versions of these theories. In Sect. 6.2 we consider two non-unit theories, namely the theory of possibly empty lists and the theory of arrays, for which superposition is known to terminate [6].

Finally we explain in Sect. 6.3 how our tool states that all these theories are combinable with $\mathcal{PC}$.

6.1 Decidability of Unit Theories

We experiment with two unit theories of lists *à la Shostak*, either without or with extensionality, and a unit theory of records. Proofs of lemmas in this section are similar to their mono-sorted counterpart given in [5], and are therefore omitted.

6.1.1 Lists without extensionality

The many-sorted unit theory of lists UL is defined by the signature $\Sigma_{UL} = \{\mathsf{car} : \text{LISTS} \to \text{ELEM}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{ELEM} \times \text{LISTS} \to \text{LISTS}\}$ and the following set of axioms:

$$\mathsf{car}(\mathsf{cons}(X, Y)) = X \tag{3}$$
$$\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y \tag{4}$$

where X is a universally quantified variable of sort ELEM and Y is a universally quantified variable of sort LISTS.

The set G_0 consists of the empty clause $\bot$ and the following constrained literals over the signature Σ_{UL}:

1. Constrained literals of sort ELEM
   a) $\mathsf{car}(l_1) = e \parallel const(l_1, e)$
   b) $e_1 =_{\text{ELEM}} e_2 \parallel const(e_1, e_2)$
   c) $e_1 \neq_{\text{ELEM}} e_2 \parallel const(e_1, e_2)$
2. Constrained literals of sort LISTS
   a) $\mathsf{cons}(e, l_1) = l_2 \parallel const(e, l_1, l_2)$
   b) $\mathsf{cdr}(l_1) = l_2 \parallel const(l_1, l_2)$
   c) $l_1 =_{\text{LISTS}} l_2 \parallel const(l_1, l_2)$
   d) $l_1 \neq_{\text{LISTS}} l_2 \parallel const(l_1, l_2)$

where e, e_1, e_2 are constrained variables of sort ELEM, l_1 and l_2 are constrained variables of sort LISTS.

The LPO ordering $>$ is used with the following requirements on the precedence of symbols: $\mathsf{cons} > \mathsf{cdr} > \mathsf{car} > l > e$ for every constant l of sort LISTS, and every constant e of sort ELEM.

Lemma 2 *The set* $\{(3), (4)\} \cup G_0$ *is saturated by* $\mathcal{SPC}$.

From an encoding of $\{(3), (4)\} \cup G_0$ our tool generates no new constrained clauses. Notice that on this example the abstraction by schematization is exact, in the following sense: the saturated set computed by $\mathcal{SPC}$ is the schematization of any saturated set computed by $\mathcal{PC}$.

6.1.2 Lists with extensionality

For this theory the signature, G_0 and the LPO ordering are the same as in Sect. 6.1.1. The set of axioms is extended with the extensionality axiom

$$\mathsf{cons}(\mathsf{car}(X), \mathsf{cdr}(X)) \quad = \quad X \tag{5}$$

where X is a universally quantified variable of sort LISTS.

Lemma 3 *The saturation of* $\{(3), (4), (5)\} \cup G_0$ *by* $\mathcal{SPC}$ *consists of* (3), (4), (5), G_0, *and the following constrained clauses:*

$$\mathsf{cons}(e, \mathsf{cdr}(l_1)) = l_2 \quad \| \quad const(e, l_1, l_2) \tag{6}$$
$$\mathsf{cons}(\mathsf{car}(l_1), l_2) = l_3 \quad \| \quad const(l_1, l_2, l_3) \tag{7}$$
$$\mathsf{car}(l_1) = \mathsf{car}(l_2) \quad \| \quad const(l_1, l_2) \tag{8}$$
$$\mathsf{cdr}(l_1) = \mathsf{cdr}(l_2) \quad \| \quad const(l_1, l_2) \tag{9}$$
$$\mathsf{cons}(\mathsf{car}(l_1), \mathsf{cdr}(l_2)) = l_3 \quad \| \quad const(l_1, l_2, l_3) \tag{10}$$

where l_1, l_2, l_3 *are constrained variables of sort* LISTS, *and* e *is a constrained variable of sort* ELEM.

Let us notice that the example given in [4] is not complete. In that paper, it is said that the saturation by $\mathcal{SPC}$ of $\{(3), (4), (5)\} \cup G_0$, consists of the constrained clauses (6) and (7), while it also contains (8), (9) and (10). From an encoding of $\{(3), (4), (5)\} \cup G_0$ our tool generates five new constrained clauses. Moreover, its trace system shows how the new constrained clauses are generated by the *Superposition* rule:

$$\begin{array}{lll}
\mathbf{sup}((3), (7)) & \mathbf{gives} & clause(\mathsf{car}(l_1) = \mathsf{car}(l_2)) \parallel const(l_1, l_2) \\
\mathbf{sup}((4), (6)) & \mathbf{gives} & clause(\mathsf{cdr}(l_1) = \mathsf{cdr}(l_2)) \parallel const(l_1, l_2) \\
\mathbf{sup}((2.c), \mathbf{sup}((5), (1.a))) & \mathbf{gives} & clause(l_2 = \mathsf{cons}(e, \mathsf{cdr}(l_1))) \parallel const(e, l_1, l_2) \\
\mathbf{sup}((2.c), \mathbf{sup}((5), (2.b))) & \mathbf{gives} & clause(l_3 = \mathsf{cons}(\mathsf{car}(l_1), l_2)) \parallel const(l_1, l_2, l_3) \\
\mathbf{sup}((2.c), \mathbf{sup}((5), (8))) & \mathbf{gives} & \\
 & & clause(l_3 = \mathsf{cons}(\mathsf{car}(l_1), \mathsf{cdr}(l_2))) \parallel const(l_1, l_2, l_3)
\end{array}$$

On this example we can see that the abstraction by schematization is an over-approximation: the abstract saturation computed by $\mathcal{SPC}$ is larger than any concrete saturation computed by $\mathcal{PC}$.

6.1.3 Records without extensionality

A record is an array with a fixed enumerated set of elements. Contrary to the theory of arrays, the theory of records can be specified by unit clauses. The termination of superposition for the theories of records with and without extensionality is shown in [6]. We consider here the theory of records

of length 3 without extensionality. It is defined by the many-sorted signature $\Sigma_{Rec} = \bigcup_{i=1}^{3}\{\mathsf{rstore}_i : \text{REC} \times \text{T}_i \to \text{REC}, \mathsf{rselect}_i : \text{REC} \to \text{T}_i\}$ and axiomatized by the following set of axioms $Ax(Rec)$:

$$\begin{array}{rcll} \mathsf{rselect}_i(\mathsf{rstore}_i(X,Y)) & = & Y & \text{for } i \in \{1,2,3\} \\ \mathsf{rselect}_i(\mathsf{rstore}_j(X,Y)) & = & \mathsf{rselect}_i(X) & \text{for } i,j \in \{1,2,3\} \text{ with } i \neq j \end{array}$$

where X is a universally quantified variable of sort REC, and Y is a universally quantified variable of sort T_i.

Let G_0 be the set composed of the empty clause $\bot$ and the following constrained literals:

1. Sort REC
   a) $\mathsf{rstore}_i(r_1, i_1) = r_2 \parallel const(r_1, r_2, i_1)$
   b) $r_1 =_{\text{REC}} r_2 \parallel const(r_1, r_2)$
   c) $r_1 \neq_{\text{REC}} r_2 \parallel const(r_1, r_2)$
2. Sort T_i $(1 \leq i \leq 3)$
   a) $\mathsf{rselect}_i(r_1) = i_1 \parallel const(r_1, i_1)$
   b) $i_1 =_{\text{T}_i} i_2 \parallel const(i_1, i_2)$
   c) $i_1 \neq_{\text{T}_i} i_2 \parallel const(i_1, i_2)$

where $i \in \{1,2,3\}$, r_1, r_2 are constrained variables of sort REC, and i_1, i_2 are constrained variables of sort T_i.

The LPO ordering $>$ is used with the following requirements on the precedence of symbols, for all $i, j \in \{1,2,3\}$: $\mathsf{rstore}_i > \mathsf{rselect}_j > r > k$ for every constant r of sort REC and every constant k of sort T_1, T_2 or T_3.

Lemma 4 *The saturation of $Ax(Rec) \cup G_0$ by $\mathcal{SPC}$ is*

$$Ax(Rec) \cup G_0 \cup \bigcup_{1 \leq i \leq 3} \{\mathsf{rselect}_i(r_1) = \mathsf{rselect}_i(r_2) \parallel const(r_1, r_2)\}.$$

From an encoding of $Ax(Rec) \cup G_0$ our tool generates the schematic saturation given in Lemma 4 which corresponds to the form of saturations described in [6]. Moreover our tool generates traces corresponding to the proof of Lemma 3 in Appendix B of [5].

6.2 Decidability of Non-unit Theories

We experiment with two non-unit theories: a theory of possibly empty lists and a theory of arrays.

6.2.1 Possibly empty lists

The signature of the theory *PEL* of possibly empty lists from [6] is composed of the unary function symbols car and cdr, the binary function symbol cons and the constant nil, denoting the empty list. For adequacy with [6], this theory is mono-sorted. The theory *PEL* is axiomatized by the following set $Ax(PEL)$ of axioms:

$$\mathsf{car}(\mathsf{cons}(X,Y)) = X \tag{11}$$
$$\mathsf{cdr}(\mathsf{cons}(X,Y)) = Y \tag{12}$$
$$\mathsf{cons}(X,Y) \neq \mathsf{nil} \tag{13}$$
$$\mathsf{cons}(\mathsf{car}(Y), \mathsf{cdr}(Y)) = Y \vee Y = \mathsf{nil} \tag{14}$$
$$\mathsf{car}(\mathsf{nil}) = \mathsf{nil} \tag{15}$$
$$\mathsf{cdr}(\mathsf{nil}) = \mathsf{nil} \tag{16}$$

where X, Y are universally quantified variables.

The set G_0 consists of the empty clause $\bot$ and the following constrained clauses:

$$x = y \quad \| \quad const(x, y) \tag{17}$$
$$x \neq y \quad \| \quad const(x, y) \tag{18}$$
$$\mathsf{car}(x) = y \quad \| \quad const(x, y) \tag{19}$$
$$\mathsf{cdr}(x) = y \quad \| \quad const(x, y) \tag{20}$$
$$\mathsf{cons}(x, y) = z \quad \| \quad const(x, y, z) \tag{21}$$

where x, y and z are constrained variables.

The LPO ordering $>$ is used with the following requirements on the precedence of symbols: $\mathsf{cons} > \mathsf{cdr} > \mathsf{car} > \mathsf{nil} > c$, where c is any constant different from nil.

Lemma 5 *The saturation of $Ax(PEL) \cup G_0$ by $\mathcal{SPC}$ consists of $Ax(PEL)$, G_0 and the following constrained clauses:*

$$\mathsf{car}(x) = y \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{22}$$
$$\mathsf{cdr}(x) = y \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{23}$$
$$\mathsf{cons}(x, y) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{24}$$
$$\mathsf{cons}(x, \mathsf{cdr}(y)) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{25}$$
$$\mathsf{cons}(\mathsf{car}(x), y) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{26}$$
$$\mathsf{car}(x) = \mathsf{car}(y) \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{27}$$
$$\mathsf{cdr}(x) = \mathsf{cdr}(y) \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{28}$$
$$\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(y)) = z \vee z = \mathsf{nil} \quad \| \quad const(x, y, z) \tag{29}$$

Proof. The set of axioms $Ax(PEL)$ is saturated. The set G_0 is also saturated. Let us now consider the union of both. *Superposition* between (11) and (25) generates a constrained clause

$$\mathsf{car}(x) = y \vee x = \mathsf{nil} \quad \| \quad const(x, y) \tag{30}$$

that can be superposed with (17) to generate the new constrained clause (22) that subsumes (30). Similarly, *Superposition* between (12) and (26) generates a constrained clause

$$\mathsf{cdr}(x) = y \vee x = \mathsf{nil} \quad \| \quad const(x, y) \tag{31}$$

whose superposition with (17) generates the new constrained clause (23) that subsumes (31). *Superposition* between (14) and (19) yields a constrained clause

$$\mathsf{cons}(x, \mathsf{cdr}(y)) = y \vee y = \mathsf{nil} \quad \| \quad const(x, y) \tag{32}$$

whose superposition with (20) generates the new constrained clause (24). The clause (32) can also be superposed with (17) to generate the new constrained clause (25) that subsumes (32). Similarly, *Superposition* between (14) and (20) yields a constrained clause

$$\mathsf{cons}(\mathsf{car}(x), y) = y \vee y = \mathsf{nil} \quad \| \quad const(x, y). \tag{33}$$

Superposition between (33) and (19) yields a renamed copy of (24), therefore subsumed by the *Subsumption* rule. A superposition between (33) and (17) generates the new constrained clause (26) that subsumes (33). *Superposition* between (11) and (21) yields a renamed copy of (19) which is immediately removed by *Subsumption*. Similarly, *Superposition* between (12) and (21) yields a renamed copy of (20) also removed by *Subsumption*. *Superposition* between (13) and (21) yields the

clause $y \neq \mathsf{nil}$, which is an elementary instance of (18) and is thus eliminated by the *Subsumption* rule. *Superposition* between (11) and (26) gives a constrained clause

$$\mathsf{car}(x) = \mathsf{car}(y) \vee y = \mathsf{nil} \quad \| \quad const(x, y) \tag{34}$$

that can be superposed with its renamed copy to produce the new constrained clause (27) that subsumes (34). Similarly, *Superposition* between (12) and (25) gives a constrained clause

$$\mathsf{cdr}(x) = \mathsf{cdr}(y) \vee y = \mathsf{nil} \quad \| \quad const(x, y) \tag{35}$$

whose superposition with its renamed copy generates the new constrained clause (28) that subsumes (35). *Superposition* between (14) and (27) gives a constrained clause

$$\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(y)) = y \vee y = \mathsf{nil} \quad \| \quad const(x, y) \tag{36}$$

whose superposition with (17) generates the new constrained clause (29) that subsumes (36). *Superposition* between (14) and (29) gives a constrained clause $x = y \vee y = \mathsf{nil} \parallel const(x, y)$ that is eliminated by the *Schematic Deletion* rule. *Superposition* between (14) and (22) gives a constrained clause $\mathsf{cons}(x, \mathsf{cdr}(y)) = y \vee y = \mathsf{nil} \vee x = \mathsf{nil} \parallel const(x, y)$ that is eliminated by *Schematic Deletion*. Similarly, *Superposition* between (14) and (23) gives a constrained clause $\mathsf{cons}(\mathsf{car}(x), y) = y \vee x = \mathsf{nil} \vee y = \mathsf{nil} \parallel const(x, y)$ that is also eliminated by *Schematic Deletion*. *Superposition* between (27) (resp. (26), (29)) and (15) generates an elementary instance of (22) (resp. (24), (25)). Similarly, *Superposition* between (28) (resp. (25), (29)) and (16) yields an elementary instance of (23) (resp. (24), (25)). All these instances are eliminated by *Subsumption*. *Superposition* between (22) and (15) and between (23) and (16) gives clauses that are composed of equalities between constrained variables and constants, that are deleted by the *Schematic Deletion* rule. All the possible applications of the *Superposition* rule between new generated clauses give clauses that are redundant with respect to the set $Ax(PEL) \cup G_0 \cup \{(22), (23), (24), (25), (26), (27), (28), (29)\}$. The *Eq. Factoring* rule cannot be applied to this set because it contains no clause satisfying the side condition of that rule. Therefore, we can conclude that the obtained set is saturated. □

Lemma 5 is consistent with the termination proof in [6] for any concrete saturation by $\mathcal{PC}$, but one can remark that descriptions of saturations slightly differ. For example, clauses of the form $\mathsf{car}(e_1) = e_2 \vee \bigvee_{i=1}^{n} c_i = d_i$ with $n \geq 1$ generated by $\mathcal{PC}$ [6] correspond to the constrained clause $\mathsf{car}(x) = y \vee z = \mathsf{nil} \parallel const(x, y, z)$ generated by $\mathcal{SPC}$.

From an encoding of $Ax(PEL) \cup G_0$ our tool generates the schematic saturation given in Lemma 5. Moreover, its trace system shows how the new constrained clauses are generated:

sup((17), **sup**((11), (25))) **gives**
$clause(\mathsf{car}(x) = y, z = \mathsf{nil}) \parallel const(x, y, z)$
sup((17), **sup**((12), (26))) **gives**
$clause(\mathsf{cdr}(x) = y, z = \mathsf{nil}) \parallel const(x, y, z)$
sup((20), **sup**((14), (19))) **gives**
$clause(\mathsf{cons}(x, y) = z, z = \mathsf{nil}) \parallel const(x, y, z)$
sup((17), **sup**((14), (19))) **gives**
$clause(\mathsf{cons}(x, \mathsf{cdr}(y)) = z, z = \mathsf{nil}) \parallel const(x, y, z)$
sup((17), **sup**((14), (20))) **gives**
$clause(\mathsf{cons}(\mathsf{car}(x), y) = z, z = \mathsf{nil}) \parallel const(x, y, z)$
sup(**sup**((11), (26)), **sup**((11), (26))) **gives**
$clause(\mathsf{car}(x) = \mathsf{car}(y), z = \mathsf{nil}) \parallel const(x, y, z)$
sup(**sup**((12), (25)), **sup**((12), (25))) **gives**
$clause(\mathsf{cdr}(x) = \mathsf{cdr}(y)z = \mathsf{nil}) \parallel const(x, y, z)$
sup((17), **sup**((14), (27))) **gives**
$clause(\mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(y)) = z, z = \mathsf{nil}) \parallel const(x, y, z)$

6.2.2 Arrays

The theory A of arrays is defined by the many-sorted signature $\Sigma_A = \{\mathsf{select} : \text{ARRAY} \times \text{INDEX} \rightarrow \text{ELEM}, \mathsf{store} : \text{ARRAY} \times \text{INDEX} \times \text{ELEM} \rightarrow \text{ARRAY}\}$ and is axiomatized by the following set $Ax(A)$ of axioms:

$$\mathsf{select}(\mathsf{store}(V, I, E), I) = E \tag{37}$$
$$\mathsf{select}(\mathsf{store}(V, I, E), J) = \mathsf{select}(V, J) \vee I = J \tag{38}$$

where V is a universally quantified variable of sort ARRAY, I, J are universally quantified variables of sort INDEX and E is a universally quantified variable of sort ELEM.

The set G_0 consists of the empty clause $\perp$ and the following constrained clauses over the signature Σ_A:

1. Sort ARRAY
   a) $\mathsf{store}(a_1, i, e) = a_2 \parallel const(a_1, i, e, a_2)$
   b) $a_1 = a_2 \parallel const(a_1, a_2)$
   c) $a_1 \neq a_2 \parallel const(a_1, a_2)$
2. Sort INDEX
   a) $i_1 = i_2 \parallel const(i_1, i_2)$
   b) $i_1 \neq i_2 \parallel const(i_1, i_2)$
3. Sort ELEM
   a) $\mathsf{select}(a, i) = e \parallel const(a, i, e)$
   b) $e_1 = e_2 \parallel const(e_1, e_2)$
   c) $e_1 \neq e_2 \parallel const(e_1, e_2)$

where a, a_1, a_2 are constrained variables of sort ARRAY, i, i_1, i_2 are constrained variables of sort INDEX and e, e_1, e_2 are constrained variables of sort ELEM.

The LPO ordering $>$ is used with the following requirements on the precedence of symbols: $a > e > i$ for all constants a of sort ARRAY, e of sort ELEM and i of sort INDEX.

Lemma 6 *The saturation of $Ax(A) \cup G_0$ by $\mathcal{SPC}$ consists of $Ax(A)$, G_0 and the following constrained clauses:*

$$\mathsf{select}(a_1, I) = \mathsf{select}(a_2, I) \vee i = I \quad \| \quad const(a_1, a_2, i) \tag{39}$$
$$\mathsf{select}(a, i) = \vee i_1 = i_2 \quad \| \quad const(a, i, e, i_1, i_2) \tag{40}$$

where i_1 and i_2 are of sort INDEX.

The proof is omitted but can be obtained and presented as the one for Lemma 5. We notice that the schematic saturation computed by $\mathcal{SPC}$ corresponds to the description given in [6] for any concrete saturation computed by $\mathcal{PC}$.

On the implementation side, from an encoding of $Ax(A) \cup G_0$ our tool generates the schematic saturation given in Lemma 6. Moreover, its trace system shows that the new constrained clauses are generated by the *Superposition* rule as follows:

sup$((38), (1.a))$ **gives**
$\quad clause(\mathsf{select}(a_1, I) = \mathsf{select}(a_2, I), i = I) \parallel const(a_1, a_2, i)$
sup$((2.a), \mathbf{sup}((39), (3.a)))$ **gives**
$\quad clause(\mathsf{select}(a, i) = e,\ i_1 = i_2) \parallel const(a, i, e, i_1, i_2)$

6.3 Combinability

This experimentation considered five theories, for lists, records and arrays. In [1, 3, 6] we can find pen-and-paper proofs that any saturation of each of these theories is finite with respect to $\mathcal{PC}$. Our implementation of schematic superposition provides mechanical proofs of these results. After computing a finite schematic saturation for each of these theories, our tool automatically checks that it does not contain any variable-active clause. Consequently, according to Lemma 1, we can conclude that all these theories are combinable with $\mathcal{PC}$.

6.4 Theory extensions of Integer Offsets

Our proof system also implements the schematic paramodulation calculus $\mathcal{SUPC}_I$ introduced in Section 4.2 to cope with theory extensions of Integer Offsets, such as LLI, the theory of lists with length given in Section 4.1. The following schematic saturations are computed by our proof system. These computations correspond to a schematic description of the forms of saturations shown in [11].

Example 1 *The schematic saturation of $Ax(LLI) \cup G_0^+$ consists of $Ax(LLI)$, G_0^+ and the following schematic literals:*

$\mathsf{s}^+(i_1) = \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$i_1 \neq \mathsf{s}^+(i_1) \parallel const(i_1, i_2)$
$\mathsf{s}^+(i_1) \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathsf{s}^+(i_1) = \mathsf{s}^+(\mathsf{len}(a)) \parallel const(i_1, a)$
$\mathsf{len}(a) = \mathsf{len}(b) \parallel const(a, b)$
$\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$

In the above schematic saturation, one can remark that $\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$ is obtained by a superposition of the constrained literal $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$ with

(a renaming of) itself. It is important to note that $\mathcal{SUPC}_I$ overapproximates $\mathcal{UPC}_I$. Hence, $\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a,b)$ has no counterpart in any saturation computed by $\mathcal{UPC}_I$, since $\mathsf{s}^m(\mathsf{len}(a)) = \mathsf{s}^n(\mathsf{len}(b))$ is simplified with $\mathcal{UPC}_I$ into either $\mathsf{len}(a) = \mathsf{s}^{n-m}(\mathsf{len}(b))$ when $m \leq n$, or $\mathsf{len}(b) = \mathsf{s}^{m-n}(\mathsf{len}(a))$ when $m > n$.

Example 2 *Let RII be the theory of records of length 3 with increment whose signature is $\Sigma_{RII} = \bigcup_{i=1}^{3}\{\mathsf{rstore}_i : \text{REC} \times \text{INT} \to \text{REC}, \mathsf{rselect}_i : \text{REC} \to \text{INT}, \mathsf{incr} : \text{REC} \to \text{REC}, \mathsf{s} : \text{INT} \to \text{INT}\}$ and whose set of axioms $Ax(RII)$ consists of $\bigcup_i^3\{\mathsf{rselect}_i(\mathsf{rstore}_i(X,Y)) = Y, \mathsf{rselect}_i(\mathsf{incr}(X)) = \mathsf{s}(\mathsf{rselect}_i(X))\}$ and $\{\mathsf{rselect}_i(\ \mathsf{rstore}_j(X,Y)) = \mathsf{rselect}_i(X)\}$ for all $i,j \in \{1,2,3\}, i \neq j$. The schematic saturation of $Ax(RII) \cup G_0^+$ consists of $Ax(RII)$, G_0^+ and the following schematic literals, for all $i \in \{1,2,3\}$:*

$\mathsf{s}^+(e_1) = \mathsf{s}^+(e_2) \parallel const(e_1,e_2)$
$e_1 \neq \mathsf{s}^+(e_2) \parallel const(e_1,e_2)$
$\mathsf{s}^+(e_1) \neq \mathsf{s}^+(e_2) \parallel const(e_1,e_2)$
$\mathsf{s}^+(\mathsf{rselect}_i(a)) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \parallel const(a,b)$
$\mathsf{s}^+(e) = \mathsf{s}^+(\mathsf{rselect}_i(a)) \parallel const(a,e)$
$\mathsf{rselect}_i(a) = \mathsf{rselect}_i(b) \parallel const(a,b)$
$\mathsf{rstore}_i(a, \mathsf{s}^+(e)) = b \parallel const(a,b,e)$

In [18], Lemma 1 has been lifted to the case of Integer Offsets: a specific condition on the form of the schematic saturation computed by $\mathcal{SUPC}_I$ is sufficient to show that a theory can be combined by using $\mathcal{UPC}_I$, along the lines of [19]. This condition is indeed satisfied for both theories LLI and RII as shown in [18], which in particular means that $\mathcal{UPC}_I$ is also a satisfiability procedure for $LLI \cup RII$.

7 Conclusion

This paper has reported on an implemented proof system for designing and verifying decision procedures. The first implementation of this proof system, based on the theoretical studies in [3, 4] has been presented in [5]. This implementation could handle only unit unsorted theories (or, equivalently, defined with only one sort), while in this paper we go further and consider non-unit theories with sorts. Thanks to our tool we can also check whether the considered theory is combinable with the paramodulation calculus.

The correctness of our tool is validated by checking the decidability of classical theories such as the theory of lists, the theory of records, the theory of possibly empty lists and the theory of arrays. For all these theories paramodulation is known to terminate. Our tool computes schematic saturations that allow us to prove termination of paramodulation in an automatic and uniform way. The schematic saturations of some theories we consider were unknown and have been discovered thanks to our tool.

Some automated deduction tools are already implemented in Maude, for instance a Church-Rosser checker [20], a coherence checker [21], etc. Our tool is a new contribution to this collection of tools. This environment will help testing new saturation strategies and experimenting new extensions of the original (schematic) superposition calculus. Since schematic superposition is interesting beyond the property of termination, we plan to extend the tool so that one can check deduction completeness and stably infiniteness [4] which are key properties for the combination of decision procedures.

Nowadays, there is a strong interest of having a native polymorphism support in automated reasoners, as advocated in [22]. The combination problem has been initiated for some polymorphic theories [23]. To build decision procedures for polymorphic theories corresponding to data structures, like for instance a polymorphic theory of arrays, it would be very natural to use a superposition-based prover enhanced with polymorphism. Hence, the problem of adding polymorphism to superposition-based provers is a very attractive research direction but it remains unexplored.

We have discussed in this paper a first extension to the case of Integer Offsets. This case has been easily implemented by reusing our tool initially developed for standard schematic paramodulation, and provides us a bunch of new interesting examples of theories with counting operators. The case of Abelian Groups [10] is more involved and would require a substantial amount of work. The reported tool support is a firm basis for further experiments in this direction.

References

[1] A. Armando, S. Ranise, M. Rusinowitch, A rewriting approach to satisfiability procedures, J. Inf. Comput 183 (2) (2003) 140 – 164.

[2] R. Nieuwenhuis, A. Rubio, Paramodulation-Based Theorem Proving, in: J. A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier and MIT Press, 2001, pp. 371–443.

[3] C. Lynch, B. Morawska, Automatic Decidability, in: Proc. of 17th IEEE Symposium on Logic in Computer Science (LICS'2002), IEEE Computer Society, Copenhagen, Denmark, 2002, pp. 7–16.

[4] C. Lynch, S. Ranise, C. Ringeissen, D.-K. Tran, Automatic Decidability and Combinability, J. Inf. Comput 209 (7) (2011) 1026–1047.

[5] E. Tushkanova, A. Giorgetti, C. Ringeissen, O. Kouchnarenko, A Rule-Based Framework for Building Superposition-Based Decision Procedures, in: F. Durán (Ed.), Proc. of 9th Int. Workshop on Rewriting Logic and Its Applications (WRLA'2012), Vol. 7571 of LNCS, Springer, Tallinn, Estonia, 2012, pp. 221–239.

[6] A. Armando, M. P. Bonacina, S. Ranise, S. Schulz, New results on rewrite-based satisfiability procedures, ACM Trans. Comput. Log. 10 (1) (2009) 1 – 51.

[7] N. Dershowitz, J.-P. Jouannaud, Rewrite Systems, in: Handbook of Theor. Comput. Sci., Volume B: Formal Models and Sematics (B), Elsevier, 1990, pp. 243–320.

[8] A. Armando, M. Bonacina, S. Ranise, S. Schulz, On a Rewriting Approach to Satisfiability Procedures: Extension, Combination of Theories and an Experimental Appraisal, in: B. Gramlich (Ed.), Proc. of the 5th Int. Workshop on Frontiers of Combining Systems (FroCoS'2005), Vol. 3717 of LNCS, Springer, Vienna, Austria, 2005, pp. 65–80.

[9] G. Godoy, R. Nieuwenhuis, Superposition with completely built-in Abelian Groups, J. Symb. Comput. 37 (1) (2004) 1–33.

[10] E. Nicolini, C. Ringeissen, M. Rusinowitch, Combinable Extensions of Abelian Groups, in: R. A. Schmidt (Ed.), Proc. of 22nd International Conference on Automated Deduction (CADE'2009), Vol. 5663 of LNCS, Springer, Montreal, Canada, 2009, pp. 51–66.

[11] E. Nicolini, C. Ringeissen, M. Rusinowitch, Satisfiability Procedures for Combination of Theories Sharing Integer Offsets, in: S. Kowalewski, A. Philippou (Eds.), Proc. of 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2009), Vol. 5505 of LNCS, Springer, York, UK, 2009, pp. 428–442.

[12] E. Tushkanova, C. Ringeissen, A. Giorgetti, O. Kouchnarenko, Automatic Decidability for Theories with Counting Operators, in: F. van Raamsdonk (Ed.), RTA'13, 2013, pp. 303–318.

[13] Audacy: Automated decidability and combinability - web interface, `http://disc.univ-fcomte.fr/audacy` (2014).

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, The Maude 2.0 System, in: R. Nieuwenhuis (Ed.), Proc. of 14th Int. Conference on Rewriting Techniques and Applications (RTA'2003), Vol. 2706 of LNCS, Springer, Valencia, Spain, 2003, pp. 76–87.

[15] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott, Unification and Narrowing in Maude 2.4, in: R. Treinen (Ed.), Proc. of 20th Int. Conference on Rewriting Techniques and Applications (RTA'2009), Vol. 5595 of LNCS, Springer, Brasília, Brazil, 2009, pp. 380–390.

[16] N. Martí-Oliet, J. Meseguer, A. Verdejo, Towards a Strategy Language for Maude, Electr. Notes Theor. Comput. Sci. 117 (2005) 417–441.

[17] S. Eker, N. Martí-Oliet, J. Meseguer, A. Verdejo, Deduction, Strategies, and Rewriting, Electr. Notes Theor. Comput. Sci. 174 (11) (2007) 3–25.

[18] E. Tushkanova, Schematic calculi for the analysis of decision procedures, Ph.D. thesis, University of Franche-Comté, Besançon, France (July 2013).

[19] C. Ringeissen, V. Senni, Modular Termination and Combinability for Superposition Modulo Counter Arithmetic, in: C. Tinelli, V. Sofronie-Stokkermans (Eds.), Proc. of 8th International Symposium on Frontiers of Combining Systems (FroCoS'2011), Vol. 6989 of LNCS, Springer, Saarbrücken, Germany, 2011, pp. 211–226.

[20] F. Durán, J. Meseguer, A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications, in: P. Ölveczky (Ed.), Proc. of 8th Int. Workshop on Rewriting Logic and Its Applications (WRLA'10), Vol. 6381 of LNCS, Springer, Paphos, Cyprus, 2010, pp. 69–85.

[21] F. Durán, J. Meseguer, A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories, in: P. Ölveczky (Ed.), Proc. of 8th Int. Workshop on Rewriting Logic and Its Applications (WRLA'10), Vol. 6381 of LNCS, Springer, Paphos, Cyprus, 2010, pp. 86–103.

[22] J. C. Blanchette, A. Paskevich, Tff1: The tptp typed first-order form with rank-1 polymorphism, in: Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA. Proceedings, Vol. 7898 of Lecture Notes in Computer Science, Springer, 2013, pp. 414–420.

[23] S. Krstic, A. Goel, J. Grundy, C. Tinelli, Combined satisfiability modulo parametric theories, in: Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of ETAPS 2007, Braga, Portugal. Proceedings, Vol. 4424 of Lecture Notes in Computer Science, Springer, 2007, pp. 602–617.