A Rule-Based Framework for Building Superposition-Based Decision Procedures

Elena Tushkanova[1,2], Alain Giorgetti[1,2], Christophe Ringeissen[1], and Olga Kouchnarenko[1,2]

[1] Inria, Villers-les-Nancy, F-54600, France
[2] CNRS FEMTO-ST and University of Franche-Comté, Besançon, F-25030, France

Abstract. This paper deals with decision procedures specified as inference systems. Among them we focus on superposition-based decision procedures. The superposition calculus is a refutation-complete inference system at the core of all equational theorem provers. In general this calculus 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 studied, for instance to automatically prove termination. This paper presents an implementation in Maude of these two inference systems. Thanks to this implementation we automatically derive termination of superposition for a couple of theories of 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 [2] allows us to build satisfiability procedures in a flexible way, by using a superposition calculus [14] (also called *Paramodulation Calculus* in [10]). 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 analysing 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 [10] to build the schematic form of the saturations. This schematic superposition calculus is very useful to analyse the behavior of the superposition calculus on a

given input theory, as shown in [9] to prove automatically the termination and the combinability of the related decision procedure.

This paper explains how to prototype the schematic superposition calculus to provide a toolkit for further experiments. The main idea is to implement the calculus so that the user can easily modify the code corresponding to an executable specification. Implementing this schematic calculus in an off-the-shelf equational theorem prover like the E prover [15] or SPASS [16] would be a difficult and less interesting task, since the developer and the user would have to understand a complex piece of code which is the result of years of engineering and debugging. To make the task easier another quite natural solution would be to use a logical framework since this calculus is defined by an inference system. This is why we propose to prototype the schematic superposition calculus by using a rule-based logical framework. Our goal is to get a rule-based program which is as close as possible to the formal specification. To achieve this goal, we propose to use Maude because Maude 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.

Our implementation of schematic superposition is very useful to get an automatic validation of saturations described in previous papers. Hence, our experimentations allow us to find a flaw in an example of [9].

The paper is structured as follows. After introducing preliminary notions and presenting superposition calculi in Section 2, Section 3 explains how we implement these calculi using the Maude system. Then Section 4 reports our experimentations with our implementation to prove the termination of superposition for theories corresponding to classical data structures such as lists and records. Section 5 concludes and presents future work.

2 Background

2.1 First-Order Logic

We assume the usual first-order syntactic notions of signature, term, position, and substitution, as defined, e.g., in [5]. We use the following notations: l, r, u, t are terms, v, w, x, y, z are variables, all other lower case letters are constant or function symbols. Given a function symbol f, a *f-rooted* term is a term whose top-symbol is f. A *compound* term is a f-rooted term for a function symbol f of arity different from 0. Given a term t and a position p, $t_{|p}$ denotes the subterm of t at position p, and $t[l]_p$ denotes the term t in which l appears as the subterm at position p. When the position p is clear from the context, we may simply write $t[l]$. The depth of a term is defined inductively as follows: $depth(t) = 0$, if t is a constant or a variable, and $depth(f(t_1, \ldots, t_n)) = 1 + max\{depth(t_i) \mid 1 \leq i \leq n\}$. A term is *flat* if its depth is 0 or 1. Application of a substitution σ to a term t (resp. a formula ψ) is written $\sigma(t)$ (resp. $\sigma(\psi)$).

A *literal* is either an equality $l = r$ or a disequality $l \neq r$. A *positive literal* is an equality and a *negative literal* is a disequality. We use the symbol $\bowtie$ to

denote either $=$ or $\neq$. The depth of a literal $l \bowtie r$ is defined as follows: $depth(l \bowtie r) = depth(l) + depth(r)$. A positive literal is *flat* if its depth is 0 or 1. A negative literal is *flat* if its depth is 0.

A first-order *formula* is built in the usual way over the universal and existential quantifiers, Boolean connectives, and symbols in a given first-order signature. We call a formula *ground* if it has no variables. A *clause* is a disjunction of literals. A *unit* clause is a clause with only one disjunct, equivalently a literal. The *empty* clause, denoted $\bot$, is the clause with no disjunct, corresponding to an unsatisfiable formula.

We also consider 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. 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. In this paper, we study decision procedures for the satisfiability problem modulo T, where $Ax(T)$ is a finite set of literals.

We consider inference systems using well-founded orderings on terms/literals that are total on ground terms/literals. An ordering $<$ on terms is a *simplification ordering* [5] if it is stable ($l < r$ implies $l\sigma < r\sigma$ 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. 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, if $t \not\leq u$, then t and u are different terms and t is maximal in $\{t, u\}$. 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. $\{l, l, r, r\}$). Also, a term is *maximal* in a literal whenever it is maximal in the corresponding multiset.

2.2 Paramodulation Calculus

In this paper we consider unit clauses, i.e. clauses composed of at most one literal. We present the restriction $\mathcal{UPC}$ (for *Unit Paramodulation Calculus*) of the inference system $\mathcal{PC}$.

Our presentation of this calculus takes the best (to our sense) from the presentations in [2], [10] and [9]. The inference system $\mathcal{UPC}$ 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. *Reflection* rule generates the empty clause when the two sides of a disequality are unifiable. Contraction rules (Fig. 2) aim at simplifying the set of literals. Using *Subsumption*, a literal 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 by *Deletion*. A fundamental feature of $\mathcal{PC}$ and $\mathcal{UPC}$ 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 [5] such that terms of positive depth are greater than constants.

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 (cf. Fig. 2). An inference is *redundant* with respect to a set S of clauses if its conclusion is redundant with respect to S. A set S of clauses 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.

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

if i) $\sigma(u) \not\leq \sigma(t)$, ii) $\sigma(l[u']) \not\leq \sigma(r)$, and iii) u' is not a variable.

Reflection $\dfrac{u' \neq u}{\bot}$

Above, u and u' are unifiable and σ is the most general unifier of u and u'.

Fig. 1. Expansion inference rules of $\mathcal{UPC}$

Subsumption $\quad \dfrac{S \cup \{L, L'\}}{S \cup \{L\}}$ **if** $L' = \sigma(L)$.

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

if i) $l' = \sigma(l)$, ii) $\sigma(l) > \sigma(r)$, and iii) $C[l'] > (\sigma(l) = \sigma(r))$.

Deletion $\quad \dfrac{S \cup \{u = u\}}{S}$

Fig. 2. Contraction inference rules of $\mathcal{UPC}$

2.3 Schematic Paramodulation Calculus

The *Schematic Unit Paramodulation Calculus* $\mathcal{SUPC}$ is an abstraction of $\mathcal{UPC}$. Indeed, any concrete saturation computed by $\mathcal{UPC}$ can be viewed as an instance of an abstract saturation computed by $\mathcal{SUPC}$, as shown by Theorem 2 in [9]. Hence, if $\mathcal{SUPC}$ halts on one given abstract input, then $\mathcal{UPC}$ halts for all the corresponding concrete inputs. More generally, $\mathcal{SUPC}$ is an automated tool to check properties of $\mathcal{UPC}$ such as termination, stable infiniteness and deduction completeness [9]. This paper focuses on termination.

$\mathcal{SUPC}$ is almost identical to $\mathcal{UPC}$, except that literals are constrained by conjunctions of atomic constraints of the form $const(x)$ where x is a variable. For sake of brevity, $const(x_1, \ldots, x_n)$ denotes the conjunction $const(x_1) \wedge \cdots \wedge const(x_n)$. $\mathcal{SUPC}$ consists of the rules in Figs. 3 and 4.

With respect to [9], we have slightly adapted the subsumption rule so that the instantiation is not only a renaming but also a substitution instantiating constrained variables by constrained variables. This allows us to have a more compact form of saturations even for simple cases, as shown in Sect. 4. For a given theory T with signature Σ, $\mathcal{SUPC}$ is executed with the input $Ax(T) \cup G_0^T$ where G_0^T is defined by

$$G_0^T = \{\bot, x = y \parallel const(x, y), x \neq y \parallel const(x, y)\} \\ \cup \bigcup_{f \in \Sigma} \{f(x_1, \ldots, x_n) = x_0 \parallel const(x_0, x_1, \ldots, x_n)\}$$

and schematizes any set of ground flat equalities and disequalities built over Σ, along with the empty clause.

2.4 Maude Language

Maude [4] is a rule-based language well-suited to implement the above 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,

Superposition $\dfrac{l[u'] \bowtie r \| \varphi \qquad u = t \| \psi}{\sigma(l[t] \bowtie r \| \varphi \wedge \psi)}$

if i) $\sigma(u) \not\leq \sigma(t)$, ii) $\sigma(l[u']) \not\leq \sigma(r)$, and iii) u' is not an unconstrained variable.

Reflection $\dfrac{u' \neq u \| \psi}{\bot}$ **if** $\sigma(\psi)$ is satisfiable.

Above, u and u' are unifiable and σ is the most general unifier of u and u'.

Fig. 3. Constrained expansion inference rules of $\mathcal{SUPC}$

Subsumption $\dfrac{S \cup \{L \| \psi, L' \| \psi'\}}{S \cup \{L \| \psi\}}$

if either a) $L \in Ax(T)$, ψ is empty and for some substitution σ, $L' = \sigma(L)$; or b) $L' = \sigma(L)$ and $\psi' = \sigma(\psi)$, 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 iv) $C[l'] > (\sigma(l) = \sigma(r))$.

Tautology $\dfrac{S \cup \{u = u \| \varphi\}}{S}$

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

Fig. 4. Contraction inference rules of $\mathcal{SUPC}$

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 nonreflective one. Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective. Narrowing [3] 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 calculi.

3 Implementation

This section describes the main ideas and principles of our implementation of $\mathcal{UPC}$ and $\mathcal{SUPC}$ in Maude.

3.1 Data Representation

Let us consider how we represent terms and literals. Maude symbols are reflected in Maude as elements of the sort `Qid` (quoted identifier). Maude terms are reflected as elements of the sorts `Constant`, `Variable` and `Term`. We exploit the Maude reflection feature by using the sort `Term` to define the new sort `Literal` for literals, as follows:

```
fmod LITERAL is
  pr META-TERM  .

  sort Literal .
  op _equals_ : Term Term -> Literal [comm] .
  op _!=_     : Term Term -> Literal [comm] .
endfm
```

The attribute `[comm]` declares that the infix binary symbols `equals` and `!=` for equality and disequality are commutative. For sets of literals we define the sort `SetLit` by instantiating the polymorphic sort `Set{X}` defined in the parameterized module `SET{X :: TRIV}` of the prelude of Maude, as follows:

```
view Literal from TRIV to LITERAL is
  sort Elt to Literal .
endv

fmod SETLIT is
  pr LITERAL .
  pr SET{Literal} * (sort Set{Literal} to SetLit) .
endfm
```

The first three lines declare that the sort `Literal` can be viewed as the sort of elements provided by the theory `TRIV`. This Maude `view` is named `Literal`. It is used in the module `SETLIT` to instantiate `Set{X}` as `Set{Literal}`. Finally, the sort `SetLit` is a renaming of the sort `Set{Literal}`. Consequently, the sets in this sort can be built by using the constant `empty`, and by using an associative, commutative, and idempotent union operator, written `_,_`. A singleton set is identified with its element (`Literal` is a subsort of `Set{Literal}`).

A schematic literal is the empty clause, an axiom, or a constrained literal. The sort `AConstr` of atomic constraints is defined by the operator

```
op const : Term -> AConstr .
```

and the sort `Constr` of constraints is a renaming of the sort `Set{AConstr}` of sets of atomic constraints. Then, the sort `SLiteral` of schematic literals is declared by

```
fmod SLITERAL is
  sort SLiteral .
  op emptyClause : -> SLiteral .
  op ax : Literal -> SLiteral .
  op _ || _ : Literal Constr -> SLiteral .
endfm
```

where the infix operator `||` constructs a constrained literal from a literal and a constraint. Similarly, for sets of schematic literals a sort `SetSLit` is defined in a module `SETSLIT`.

3.2 Inference Rules

This section presents the encoding of $\mathcal{SUPC}$, the encoding of $\mathcal{UPC}$ being similar. 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. More precisely, the encoding description starts with the translation of some contraction rules into rewrite rules (the simplification rule is omitted). Afterwards, 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.

Contraction rules The following Maude conditional rewrite rule encodes the first case of *Subsumption* inference rule in $\mathcal{SUPC}$:

```
crl [subsum1] : (ax(L1), (L2 || Phi2)) => ax(L1)
  if LiteralMatch(L1, L2) =/= noMatch .
```

The function call `LiteralMatch(L1, L2)` checks if the second literal `L2` is matched by the first one (`L1`), by calling the Maude function `metaMatch`.

The following two Maude conditional rewrite rules encode the second case of *Subsumption* inference rule, decomposed into two cases:

```
crl [subsum2] : L1 || Phi1, L2 || Phi2 => L1 || Phi1
  if isRename(L1 || Phi1, L2 || Phi2) .

crl [subsum3] : L1 || Phi1, L2 || Phi2 => L1 || Phi1
  if filter(L1 || Phi1, L2 || Phi2) .
```

The function `isRename` checks if one constrained literal is the renaming of another one by checking the existence of a substitution mapping the first literal into the second one, and the constraint of the first literal into the constraint of the second one. Moreover, this substitution should replace variables by variables and the correspondence between the replaced variables and the replacing ones should be one to one. The function call `filter(L1 || Phi1, L2 || Phi2)` checks if the constrained literal `L1 || Phi1` is more general than the constrained literal `L2 || Phi2` by determining the existence of a substitution mapping the

first literal into the second one, and the constraint of the first literal into the constraint of the second one.

The *Simplification* inference rule rewrites a literal into a simpler one by using an axiom as a rewrite rule. This is performed by the Maude function `metaFrewrite` that rewrites the metarepresentation of a term with the rules defined in the metarepresentation of a module. In our implementation, a function `addRl` adds an axiom to the metarepresentation of a module `INITIAL-MODULE` where all the functional symbols are defined.

```
op addRl : Term Term -> Module .
eq addRl(L, R) = addRules(
   (rl (L) => (R) [none] .), upModule('INITIAL-MODULE, false)) .
```

This function uses the Full Maude function `addRules` that takes a set of rules and a module as parameters. The axiom `ax(L equals R)` is added by the function call `addRl(L,R)`.

The inference rule *Tautology* is simply encoded by the rewrite rule

```
rl [tautology] : U equals U || Phi => empty .
```

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

```
crl [del] : L || Phi => empty if isSatisfiable(Phi) == false .
```

where the function `isSatisfiable` checks if a given constraint holds, i.e. none of the terms it constraints is compound.

Expansion rules 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 [11, 8], but unfortunately it appeared not to be compatible with the Full Maude version 2.5b required for narrowing (see details below).

To circumvent this technical problem we propose to control rules with states. We consider three distinct states, for the sets of literals derived by $\mathcal{SUPC}$. These states and the sort of states are defined as follows:

```
mod STATE is
  pr SETSLIT .
  sort State .
  op state : SetSLit -> State .
  op _selectOneLitFromGenSet_ : SetSLit SetSLit -> State .
  op _redundancy_ : SetSLit SLiteral -> State .
endm
```

The input state of the expansion rules of $\mathcal{SUPC}$ is expected to be of the form `state(`S`)` where S is a set of schematic literals.

The *Reflection* rule checks whether a given set of schematic literals contains a constrained disequality whose two sides are unifiable by a substitution that also satisfies the constraint. In this case the empty clause is added to the set of literals. The *Reflection* rule is encoded by the following conditional rewrite rule:

```
crl [reflection] :
  state((S, U' != U || Phi)) =>
    state((S, U' != U || Phi, emptyClause))
    if isSatisf(U' != U || Phi) .
```

where the function `isSatisf` performs the above mentioned checking.

The *Superposition* rule

$$\frac{l[u'] \bowtie r \| \varphi \qquad u = t \| \psi}{\sigma(l[t] \bowtie r \| \varphi \wedge \psi)}$$

produces a new literal of the form $\sigma(l[t] \bowtie r\|\varphi \wedge \psi)$ from any set containing two schematic literals (axioms or constrained literals) of the form $l[u'] \bowtie r\|\varphi$ and $u = t\|\psi$, 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 second literal $u = t$ as a rewriting rule $u \to t$, to narrow the left-hand side $l[u']$ of the first literal. If the narrowing succeeds it produces a term $\sigma(l[t])$ where σ is a most general unifier of u and u'. It remains to apply σ to the right-hand side r of the first literal and to the conjunction of the two constraints φ and ψ.

To narrow we use a function `metaENarrowShowAll` already implemented in Full Maude version 2.5. In this version the narrowing was restricted to non-variable positions, along its standard definition. But the *Superposition* rule of $\mathcal{SUPC}$ requires the unusual feature: narrowing should also be applied at the positions of the variables schematizing constants. Therefore we have asked Santiago Escobar, the developer of narrowing in Full Maude, to implement this feature. As an answer to this request, he has introduced a flag `alsoAtVarPosition` to the narrowing function for disabling the standard restriction.

A second difficulty is that the `metaENarrowShowAll` function called on 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 produces only one literal. To solve this problem two additional states S `selectOneLitFromGenSet` S' and S `redundancy` L have been introduced, where S is a given set of schematic literals, S' is the set of schematic literals produced by the narrowing function applied to two schematic literals from S, and L is one schematic literal. Then *Superposition* is encoded by four Maude rewriting rules named `sup`, `select`, `no-sup` and `pick`. The `sup` rule is defined by

```
rl [sup] : state((S, L1, L2)) =>
  (S, L1, L2) selectOneLitFromGenSet applySup(L1, L2) .
```

where the function `applySup` generates from `L1` and `L2` a set of new schematic literals by calling the narrowing function and checking the ordering conditions of the *Superposition* rule. The ordering conditions invoke a function implementing the orderings detailed in Section 3.4. When the set of new schematic literals is empty, the rule

```
rl [no-sup] : S selectOneLitFromGenSet empty => state(S) .
```

returns the input set in a state ready for another expansion. Otherwise, the rule

```
rl [select] : S selectOneLitFromGenSet (L, S') =>
  if checkConstr(L) then S redundancy L
    else S selectOneLitFromGenSet S' fi .
```

considers one by one the schematic literals in the new set until the set is empty or a schematic literal `L` with a satisfiable constraint is found. Satisfiability is checked by invoking the function `checkConstr`. If the constraint of `L` is satisfiable then a state `S redundancy L` is constructed. It is an input state for the rule

```
rl [pick] : S redundancy L =>
     if L isRedundant S == false
        then state((S, L))
        else state(S)
     fi .
```

which checks if a generated schematic literal `L` is redundant with respect to a given set `S` of schematic literals. The redundancy is checked by the function `isRedundant` that uses the Maude function `metaSearch`. This function tries to reach the set `S` from the union `(S, L)` of `S` and {`L`} by applying contraction rules. If the new schematic literal is not redundant then it is added to the state, otherwise, the state is unchanged.

3.3 Saturation

A forward search for generated sets of schematic literals is performed by a function `searchState` defined by

```
op searchState : State Nat -> State .
eq searchState(S', N) = downTerm(getTerm(metaSearch(
  upModule('SP, false), upTerm(S'), 'state['S:SetSLit],
  nil, '*, unbounded, N)), error1) .
```

where `SP` is a module where all the expansion rules are defined. The function call `searchState(S,N)` tries to reach the `N`th state from an initial state `S` by applying the expansion rules. It uses a breadth-first exploration of the reachable state space, which is a fundamental graph traversal strategy implemented by the Maude `metaSearch` function with the `'*` parameter. When the Maude function `downTerm` fails in moving down the meta-represented term given as its first argument, it returns its second argument, namely `error1`, which is declared as a constant of sort `State` (`op error1 : -> [State] .`).

Then the principle of saturation is implemented by the function `saturate` defined by

```
op saturate : State -> State .
eq saturate(St) =
  if searchState(St, 1) == error1 then St else
    if searchState(St, 1) =/= St then saturate(searchState(St, 1))
    else St fi fi .
```

which implements a fixpoint algorithm in order to get the state of a saturated set of schematic literals. If the initial state is already saturated, then the function returns it unchanged.

A saturated set of schematic literals 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 intermediary states is also interesting for debugging purposes. Note that the Maude `metaSearch` function is already used with the parameter `'!` to apply contraction rules.

3.4 Orderings

A fundamental feature of our superposition calculi is the usage of a simplification ordering which is total on ground terms. This section presents all the orderings used in the side conditions of the inference rules and describes their implementation. In our calculi, 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 1. *Given a precedence $>_F$ on function symbols, the* ***lexicographic path ordering (LPO)*** *$>_{lpo}$ [5] is defined as follows:*

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

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

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

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

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 $s_1, \ldots, s_n, t_1, \ldots, t_m, u_1, \ldots, u_p, t$ 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 lexicographic extension can be specified as an inference system that can be directly encoded in Maude.

The LPO ordering is implemented as a Boolean function `gtLPO()` such that $\texttt{gtLPO}(s, \texttt{SC}, t) = \texttt{true}$ if and only if $s >_{lpo} t$. One can remark the additional parameter `SC`. It collects the constrained variables that are viewed as constants in the precedence ordering: constrained variables are smaller than non-constant function symbols.

Let us briefly present the four main rules implementing $\texttt{gtLPO}(s, \texttt{SC}, t)$.

1. When $n \geq 1$ and $m \geq 1$, the rule *LPO1* is encoded by

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

   where `NeSL` and `NeTL` are non-empty lists of terms. Here the head symbols of s and t are equal. Then the list of subterms `NeSL` of $s = \mathtt{F}[\mathtt{NeSL}]$ 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.
2. When $n \geq 1$ and $m \geq 1$, the rule *LPO2* is encoded by

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

   Here the heads of s and t are not equal. Then the head of s should be greater than the head of t and s should be greater than all the direct subterms of t.
3. The rule *LPO3* is encoded by

```
ceq gtLPO(F[UL1, Uk, UL2], SC, t) = true
  if gtLPO(Uk, SC, t) == true .
```

   whose condition checks whether a direct subterm of $s = \mathtt{F}[\mathtt{UL1}, \mathtt{Uk}, \mathtt{UL2}]$ is greater than t.
4. The rule

```
eq gtLPO(F[UL1, Uk, UL2], SC, Uk) = true .
```

   encodes *LPO4*, when a direct subterm of $s = \mathtt{F}[\mathtt{UL1}, \mathtt{Uk}, \mathtt{UL2}]$ is equal to 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 $\{l, l, r, r\}$. As for the lexicographic extension, the multiset extension can be specified as an inference system that can be directly encoded in Maude.

4 Experimentations

We have done some experiments to compare the (schematic) saturations computed by our tool with corresponding results we can find in the literature. For the theory of lists without extensionality, our tool generates the same saturation as the one given in [10]. More surprisingly, for the theory of lists with extensionality, our implementation reveals that the description given in [9] for the saturation is incomplete. We also consider the case of records of length 3 for which superposition is known to terminate on ground literals [1].

4.1 Theories of Lists

We experiment with two theories of lists *à la Shostak*, either without or with extensionality.

Let $\Sigma_{List} = \{cons, car, cdr\}$ be the signature of the theory of lists. The set G_0^{List} consists of the empty clause $\bot$ and the following schemas of ground flat literals over the signature Σ_{List}:

$$x = y \parallel const(x, y) \tag{1}$$
$$x \neq y \parallel const(x, y) \tag{2}$$
$$car(x) = y \parallel const(x, y) \tag{3}$$
$$cdr(x) = y \parallel const(x, y) \tag{4}$$
$$cons(x, y) = z \parallel const(x, y, z) \tag{5}$$

where x, y and z are constrained variables.

Theory of lists without extensionality The theory of lists without extensionality is axiomatized by the following two axioms:

$$car(cons(X, Y)) = X \tag{6}$$
$$cdr(cons(X, Y)) = Y \tag{7}$$

where X and Y are universally quantified variables.

Lemma 1. *The set $G_0^{List} \cup \{(6), (7)\}$ is saturated by $\mathcal{SUPC}$.*

This result is given in [10]. The interested reader can find our proof in Appendix A.

From an encoding of $G_0^{List} \cup \{(6), (7)\}$ our tool generates no new schematic literal. Notice that on this example the abstraction by schematization is exact, in the following sense: the saturated set computed by $\mathcal{SUPC}$ is the schematization of any saturated set computed by $\mathcal{UPC}$.

Theory of lists with extensionality This theory is axiomatized by the two axioms (6) and (7), plus the axiom (called the extensionality axiom)

$$cons(car(X), cdr(X)) = X \tag{8}$$

where X is a universally quantified variable.

Lemma 2. *The saturation of $G_0^{List} \cup \{(6), (7), (8)\}$ by $\mathcal{SUPC}$ consists of G_0^{List}, (6), (7), (8) and the following constrained literals:*

$$cons(x, cdr(y)) = z \parallel const(x, y, z) \tag{9}$$
$$cons(car(x), y) = z \parallel const(x, y, z) \tag{10}$$
$$car(x) = car(y) \parallel const(x, y) \tag{11}$$
$$cdr(x) = cdr(y) \parallel const(x, y) \tag{12}$$
$$cons(car(x), cdr(y)) = z \parallel const(x, y, z) \tag{13}$$

Proof. The set of axioms $\{(6), (7), (8)\}$ is saturated. The set G_0^{List} is also saturated. It remains to show the same property for the union of both.

Superposition between (6) and (5) and between (7) and (5) respectively yields renamings of (3) and (4), which are immediately removed by the subsumption rule. *Superposition* between (8) and (3) yields the new constrained literal

$$cons(x, cdr(y)) = y \parallel const(x, y). \tag{14}$$

Then, *Superposition* between (14) and (1) gives the constrained literal (9), which subsumes (14). Similarly, *Superposition* between (8) and (4) yields the new constrained literal

$$cons(car(x), y) = x \parallel const(x, y) \tag{15}$$

and *Superposition* between (15) and (1) gives the constrained literal (10), which subsumes (15). *Superposition* between (6) and (10) and between (7) and (9) respectively gives the constrained literals (11) and (12). *Superposition* between (8) and (11) gives the new constrained literal

$$cons(car(x), cdr(y)) = y \parallel const(x, y) \tag{16}$$

and *Superposition* between (16) and (12) gives the constrained literal (13), which subsumes (16). *Superposition* between any axiom and (1) yields constrained literals that are immediately removed by the subsumption rule. Any other application of *Superposition* rule between an axiom and a constrained literal yields a constrained literal that is already in the set $G_0^{List} \cup \{(6), (7), (8)\} \cup \{(9), (10), (11), (12), (13)\}$. Since no other rule can be applied to this set of schematic literals, we conclude that it is saturated. □

The example given in [9] is not complete. In that paper, it is said that the saturation by $\mathcal{SUPC}$ of $G_0^{List} \cup \{(6), (7), (8)\}$, consists of the constrained literals (9) and (10), while it also contains (11), (12) and (13). From an encoding of $G_0^{List} \cup \{(6),\ (7), (8)\}$ our tool generates these five new constrained literals. On this example we can see that the abstraction by schematization is a over-approximation: the abstract saturation computed by $\mathcal{SUPC}$ is larger than any concrete saturation computed by $\mathcal{UPC}$.

4.2 Theory of Records

A record can be considered as a special form of array where the number of elements is fixed. 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 [1]. We consider here the theory of records of length 3 without extensionality given by the signature $\Sigma_{Rec} = \bigcup_{i=1}^{3} \{rstore_i, rselect_i\}$ and axiomatized by the following set of axioms $Ax(Rec)$:

$$rselect_i(rstore_i(X, Y)) = Y \text{ for all } i \in \{1, 2, 3\}$$

and

$$rselect_i(rstore_j(X,Y)) = rselect_i(X,Y) \text{ for all } i,j \in \{1,2,3\},\ i \neq j,$$

where X and Y are universally quantified variables. Let G_0^{Rec} be defined as in Section 2.3.

Lemma 3. *The saturation of $G_0^{Rec} \cup Ax(Rec)$ by $\mathcal{SUPC}$ consists of G_0^{Rec}, $Ax(Rec)$ and the constrained literals*

$$rselect_i(x) = rselect_i(y) \quad \| \quad const(x,y)$$

for $i = 1,2,3$.

A proof of this lemma can be found in Appendix B. From an encoding of $G_0^{Rec} \cup Ax(Rec)$ our tool generates the schematic saturation given in Lemma 3 which corresponds to the form of saturations described in [1].

5 Conclusion

This paper reported on a prototyping environment for designing and verifying decision procedures. This environment, based on the theoretical studies in [10, 9], is the first implementation including both superposition and schematic superposition calculi. It has been implemented from scratch on the firm basis provided by Maude. Some automated deduction tools are already implemented in Maude, for instance a Church-Rosser checker [6], a coherence checker [7], 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. A short term future work is to consider non-unit clauses. Since schematic superposition is interesting beyond the property of termination, we also want to extend the implementation so that we can check deduction completeness and stably infiniteness [9] which are key properties for the combination of decision procedures. We are also interested in developing new schematic calculi for superposition modulo fragments of arithmetic such as Integer Offsets [13] and Abelian Groups [12]. The reported implementation is a firm basis for all these future developments.

Acknowledgments. We are deeply grateful to Santiago Escobar for his help on the use of narrowing in Maude and to Alberto Verdejo for his answers about the strategy language for Maude.

References

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

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

[3] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., L. Talcott, C.: Unification and narrowing in Maude 2.4. In: Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, Proceedings. pp. 380–390 (2009)
[4] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., F. Quesada, J.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science (2001)
[5] Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Formal Models and Sematics, vol. B, pp. 243–320. MIT Press (1990)
[6] Durán, F., Meseguer, J.: A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In: Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6381, pp. 69–85. Springer (2010)
[7] Durán, F., Meseguer, J.: A Maude coherence checker tool for conditional order-sorted rewrite theories. In: Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6381, pp. 86–103. Springer (2010)
[8] Eker, S., Martí-Oliet, N., Meseguer, J.and Verdejo, A.: Deduction, strategies, and rewriting. Electr. Notes Theor. Comput. Sci. 174(11), 3–25 (2007)
[9] Lynch, C., Ranise, S., Ringeissen, C., Tran, D.K.: Automatic decidability and combinability. Inf. Comput. 209(7), 1026–1047 (2011)
[10] Lynch, C., Morawska, B.: Automatic decidability. In: Proc. of 17th IEEE Symposium on Logic in Computer Science, (LICS'02), Copenhagen, Denmark. pp. 7–16. IEEE Computer Society Press (2002)
[11] Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. Electr. Notes Theor. Comput. Sci. 117, 417–441 (2005)
[12] Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combinable extensions of Abelian groups. In: Schmidt, R. (ed.) Proc. of 22nd International Conference on Automated Deduction, (CADE'09). LNAI, vol. 5663, pp. 51–66. Springer, Montreal (Canada) (2009)
[13] Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combining satisfiability procedures for unions of theories with a shared counting operator. Fundamenta Informaticae 105(1-2), 163–187 (2010)
[14] Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 371–443. Elsevier and MIT Press (2001)
[15] Schulz, S.: E - a brainiac theorem prover. AI Commun. 15(2-3), 111–126 (2002)
[16] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R. (ed.) Proc. of 22nd International Conference on Automated Deduction, (CADE'09). LNAI, vol. 5663, pp. 140–145. Springer, Montreal (Canada) (2009)

A Schematic Saturation of Lists

Lemma 1. *Let*

$$G_0^{List} = \{\bot\} \cup \begin{cases} x = y & \| \ const(x,y) & (1) \\ x \neq y & \| \ const(x,y) & (2) \\ car(x) = y & \| \ const(x,y) & (3) \\ cdr(x) = y & \| \ const(x,y) & (4) \\ cons(x,y) = z & \| \ const(x,y,z) & (5) \end{cases}$$

and let

$$Ax(List) = \begin{cases} car(cons(X,Y)) = X & (6) \\ cdr(cons(X,Y)) = Y & (7) \end{cases}$$

The set $G_0^{List} \cup Ax(List)$ is saturated by $\mathcal{SUPC}$.

Proof. The set of axioms $\{(6),(7)\}$ is saturated. The set G_0^{List} is also saturated. It remains to show the same property for the union of both.

Superposition between (6) and (5) yields a renaming of (3), which is immediately removed by the subsumption rule. Similarly, *Superposition* between (7) and (5) yields a renaming of (4), which is removed by the subsumption rule as well. *Superposition* between any axiom and (1) yields a schematic literals that are immediately removed by the subsumption rule. Since no other rule can be applied between an axiom and a schematic literal, we conclude that the set $G_0^{List} \cup \{(6),(7)\}$ is saturated. □

B Schematic Saturation of Records

Lemma 3. *Let G_0^{Rec} be the set that consists of the empty clause $\bot$ and the constrained literals*

$$x = y \ \| \ const(x,y) \quad (17)$$
$$x \neq y \ \| \ const(x,y) \quad (18)$$
$$rstore_1(x,y) = z \ \| \ const(x,y,z) \quad (19)$$
$$rstore_2(x,y) = z \ \| \ const(x,y,z) \quad (20)$$
$$rstore_3(x,y) = z \ \| \ const(x,y,z) \quad (21)$$
$$rselect_1(x) = y \ \| \ const(x,y) \quad (22)$$
$$rselect_2(x) = y \ \| \ const(x,y) \quad (23)$$
$$rselect_3(x) = y \ \| \ const(x,y) \quad (24)$$

Let $Ax(Rec)$ be the set of axioms

$$rselect_1(rstore_1(X,Y)) = Y \quad (25)$$
$$rselect_2(rstore_2(X,Y)) = Y \quad (26)$$

$$rselect_3(rstore_3(X, Y)) = Y \tag{27}$$
$$rselect_1(rstore_2(X, Y)) = rselect_1(X) \tag{28}$$
$$rselect_1(rstore_3(X, Y)) = rselect_1(X) \tag{29}$$
$$rselect_2(rstore_1(X, Y)) = rselect_2(X) \tag{30}$$
$$rselect_2(rstore_3(X, Y)) = rselect_2(X) \tag{31}$$
$$rselect_3(rstore_1(X, Y)) = rselect_3(X) \tag{32}$$
$$rselect_3(rstore_2(X, Y)) = rselect_3(X) \tag{33}$$

The saturation of $G_0^{Rec} \cup Ax(Rec)$ *by* $\mathcal{SUPC}$ *consists of* G_0^{Rec}, $Ax(Rec)$ *and the following constrained literals:*

$$rselect_1(x) = rselect_1(y) \parallel const(x, y) \tag{34}$$
$$rselect_2(x) = rselect_2(y) \parallel const(x, y) \tag{35}$$
$$rselect_3(x) = rselect_3(y) \parallel const(x, y) \tag{36}$$

Proof. The set of axioms $Ax(Rec)$ is saturated. The set of schematic literals G_0^{Rec} is also saturated. It remains to show the same property for the union of both.

Superposition between (25) and (19) yields a renaming of (22), which is immediately removed by the subsumption rule. It is similar for the indices 2 and 3, between (26) and (20) and between (27) and (21).

Superposition between (28) and (20) yields the constrained literal (34). Afterwards, *Superposition* between (29) and (21) yields a renaming of (34), which is immediately removed by the subsumption rule. It is similar for the indices 2 and 3, between (30) and (19) and between (32) and (19).

Superposition between any axiom and (17) yields schematic literals that are immediately removed by the subsumption rule. Since no other rule can be applied between an axiom and a schematic literal, we conclude that the set $G_0^{Rec} \cup Ax(Rec) \cup \{(34), (35), (36)\}$ is saturated for $\mathcal{SUPC}$. □