Prime Implicate Generation in Equational Logic (extended abstract)*

Mnacho Echenim1, Nicolas Peltier1, Sophie Tourret2
1 CNRS, LIG, University of Grenoble Alpes, Grenoble, France
2 Max Planck Institute for Informatics, Saarbrücken

Abstract
A procedure is proposed to efficiently generate sets of ground implicates of first-order formulas with equality. It is based on a tuning of the superposition calculus [Nieuwenhuis and Rubio, 2001], enriched with rules that add new hypotheses on demand during the proof search. Experimental results are presented, showing that the proposed approach is more efficient than state-of-the-art systems.

1 Introduction and Motivations
Abductive reasoning is the process of inferring, given a set of axioms A and a formula φ, a set of assertions H such that A ∪ H ⊢ φ. The set H may be viewed as a set of hypotheses that are sufficient to ensure the validity of the entailment A ⊢ φ, or as a set of explanations of φ. Such hypotheses must be economical and plausible, in particular H must be minimal w.r.t. logical entailment and A ∪ H must be satisfiable. Abductive reasoning has many applications in artificial intelligence, verification, and debugging, e.g. for computing missing pre-conditions, spotting and correcting errors in a logical specification, or for dealing with approximative, incomplete or spurious information. The problem has been thoroughly investigated in propositional logic [Marquis, 2000], and very efficient algorithms have been proposed [Simon and Del Val, 2001; Previti et al., 2015], but only a few approaches handle more expressive logics [Knill et al., 1993; Marquis, 1991; Mayer and Pirri, 1993; Nabeshima et al., 2010]. In particular, none of them is able to deal with the equality predicate in an efficient way.

In the present paper, we tackle the problem of generating such a set H, when A ∪ {φ} is a set of first-order formulas with equality and H is a set of ground unit clauses. In this case, by duality, ¬H is a clausal logical consequence of A ∪ {¬φ}, i.e., an implicate of A ∪ {¬φ}, and the problems boils down to efficiently generating sets of (entailment-minimal) implicates or prime implicates.

We illustrate our approach by an example in verification. Consider the following toy program, redirecting the tail of a nonempty list l1 to a list l2 of length 1. Assume we want to check that l1 is of length 2 after the program is executed. This post-condition does not hold, which may come as a surprise for some programmers. Indeed, if l1 = l2, then the tail of l1 is redirected to l1 yielding a cyclic list (hence length(l1) is either 1 or undefined, depending on the definition). The problem can be stated in first-order logic with equality as the following set of clauses S, where the tail function is represented by a function tail : heap × list → list, and where h, h′ : heap are constant symbols denoting the initial and final states of the heap and x, y : list, z : heap are universally quantified variables.

\begin{algorithm}
\textbf{Example(List l1, List l2)}
\begin{itemize}
  \item \textbf{requires} length(l1) \neq 0;
  \item \textbf{requires} length(l2) = 1;
  \item let l1, tail = l2;
  \item \textbf{ensures} length(l1) = 2;
\end{itemize}
\end{algorithm}

% Definition of the tail redirection operation
% The tail of x is redirected to y:
\begin{align*}
x \simeq \text{nil} & \lor \text{tail}(\text{SetTail}(z, x, y), x) \simeq y \\
% The tail of the other lists is not affected by the redirection:
\end{align*}
\begin{align*}
x \simeq \text{nil} & \lor x' \simeq \text{nil} \lor x' \simeq x \\
\lor \text{tail}(\text{SetTail}(z, x, y), x') & \simeq \text{tail}(z, x') \\
% Definition of length
\end{align*}
\begin{align*}
length(z, \text{nil}) & \simeq 0 \\
x \simeq \text{nil} & \lor length(z, x) \simeq s(length(z, \text{tail}(z, x))) \\
% Pre-conditions
\end{align*}
\begin{align*}
length(h, l_1) & \neq 0 \\
length(h, l_2) & \simeq s(0) \\
% Negation of the post-condition
\end{align*}
\begin{align*}
length(h', l_1) & \neq s(s(0)) \\
% Translation of the program
\end{align*}
\begin{align*}
h' = \text{SetTail}(h, l_1, l_2)
\end{align*}

The set S is satisfiable, the problem is to show that S is unsatisfiable under the hypothesis l1 \neq l2, or equivalently that l1 \simeq l2 is an implicate of S.

2 Normalized Ground Clauses
Terms, atoms, literals and clauses are defined inductively as usual over a set of function symbols Σ and variables V. We

\textsuperscript{*}This paper is an extended abstract of an article in the Journal of Artificial Intelligence Research [Echenim et al., 2017].
assume, w.l.o.g., that the equality \( \equiv \) is the only predicate symbol. We use the symbol \( \bowtie \) to denote either \( \equiv \) or \( \not\equiv \). Substitutions are functions mapping variables to terms, extended to any expression (terms, atoms, clauses, etc.) in the natural way and written in postfix notation. We assume given a reduction order \( \prec \), i.e., an order among terms that is closed under substitution and context embedding, and contains the subterm ordering (this entails that \( \prec \) is well-founded). The order \( \prec \) is extended to atoms or literals by interpreting them as multisets of terms, and to clauses by the usual multiset extension. For any expression \( E \), we denote by \( E^c \) the nnf of the negation of \( E \), formally defined as follows: \( (s \equiv t)^c = s \not\equiv t, (s \not\equiv t)^c = t \equiv s \), \( \bigwedge_{i=1}^n t_i^c = \bigwedge_{i=1}^n l_i^c \) and \( \bigvee_{i=1}^n t_i^c = \bigvee_{i=1}^n l_i^c \).

**Definition 1** Given a clause \( C \), we define the relation \( \equiv_C \) on terms as follows: for all terms \( s, t, s \equiv_C t \) iff \( C^c \models s \equiv t \). The \( C \)-representatives of a term \( s \), literal \( l \) and clause \( l_1 \lor \cdots \lor l_n \) are respectively defined by:

\[
\begin{align*}
\defn l_1 \lor \cdots \lor l_n &\defn \left( l_1 \lor \cdots \lor l_n \right) \left( l_1 \lor \cdots \lor l_n \right) \\
(s \bowtie t) &\defn \min_{s \mid t} \{ s \mid t \equiv_C s \} \\
(c \bowtie t)_C &\defn s \bowtie t \lor \left( l_1 \lor \cdots \lor l_n \right) \\
(l_1 \lor \cdots \lor l_n)_C &\defn l_1 \lor \cdots \lor l_n \\
\end{align*}
\]

**Definition 2** A ground clause \( C \) is normalized if:

1. every literal \( l \) in \( C \) is such that \( l \neq \bot \); and
2. there are no two distinct positive literals \( l, m \) in \( C \) such that \( m \bowtie_{C \bowtie} \bot \) is a tautology;
3. \( C \) contains no literal of the form \( \bot \).

A conjunction of literals \( X \) is normalized if the clause \( X^c \) is normalized.

**Proposition 3** Any ground falsifiable clause \( C \) admits a unique equivalent normalized clause \( C_1 \), which is the \( \bowtie \)-smallest clause equivalent to \( C \). Consequently, two clauses are equivalent iff either they are both tautological or they have the same normalized form.

### 3 A Constrained Superposition Calculus

This section contains the definition of the calculus for generating implicates, named \( cSP \). It is based on the Superposition Calculus \( SP \) (see for instance [Nieuwenhuis and Rubio, 2001]) which is the most successful proof procedure for first-order logic with equality. The principle underlying \( cSP \) consists in applying the inference rules of \( SP \) to the set of clauses under consideration along with ground unit clauses that are added during proof search and that act as hypotheses. To keep track of the hypotheses that were used to derive a clause, the former are attached to the clauses as constraints. This yields the following definition:

**Definition 4** A constraint is a conjunction (or set) of ground literals. The empty (tautological) constraint is denoted by \( \top \).

A constrained clause (or c-clause) is a pair \( [C \mid X] \) where \( C \) is a clause and \( X \) is a constraint. The c-clause \( [C \mid \top] \) is simply represented as \( C \), and referred to as a standard clause.

Intuitively \( X \) denotes the set of hypotheses used to derive \( C \). If \( C \) is empty, then \( [C \mid X] \) is equivalent to \( X^c \), and \( X^c \) is an implicite of the clause set under consideration.

<table>
<thead>
<tr>
<th>Inference Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>c-Superposition</td>
<td>([t \equiv s \lor C \mid X] [u[t] \bowtie v \lor D \mid \gamma] ]</td>
</tr>
<tr>
<td>(i), (ii), (iii)</td>
<td>([u[s] \bowtie v \lor C \lor D \mid \gamma] ]</td>
</tr>
<tr>
<td>c-Factoring</td>
<td>([t \equiv u \lor t' \equiv v \lor C \mid X] ]</td>
</tr>
<tr>
<td>(iv), (ix)</td>
<td>([t \equiv u \lor v \equiv t' \lor C \mid X] ]</td>
</tr>
<tr>
<td>c-Reflection</td>
<td>([t \equiv t' \lor C \mid X] ]</td>
</tr>
<tr>
<td>(v), (ix)</td>
<td>([C \mid X] ]</td>
</tr>
</tbody>
</table>

**Positive Assertion**

\([u[t] \bowtie v \lor C \mid X] \] where the following conditions hold:

For all rules: \( \sigma = \text{mgu}(t, t') \);

(i): \( (u[t'] \bowtie v) \sigma \in \text{sel}((u[t'] \bowtie v \lor D) \sigma) \) and \( v \sigma \not\equiv u[t'] \sigma \);
(ii): \( (t \equiv s) \sigma \in \text{sel}((t \equiv s \lor C) \sigma) \) and \( s \sigma \not\equiv t \sigma \);
(iii): \( X \lor Y \not\equiv X \); or
(iv): \( (t \equiv u) \sigma \in \text{sel}((C \lor t \equiv u \lor t' \equiv v) \sigma) \);
(v): \( (t \equiv t') \sigma \in \text{sel}((C \lor t \equiv t' \lor C) \sigma) \);
(vi): \( s \equiv t', v \sigma \not\equiv u[t'] \sigma \) and \( (u[t] \bowtie v) \sigma \in \text{sel}((u[t] \bowtie v \lor v \lor C) \sigma) \);
(vii): \( X \lor t' \equiv s \not\equiv X \);
(viii): \( v \not\equiv u[t'] \);
(ix): \( X \not\equiv X \), or
(x): \( X \lor u[t'] \bowtie v \not\equiv v \in X \).

The set of inference rules defining \( cSP \) are given in Figure 1. The notation \( u[t] \) is used to denote a term containing a subterm \( t \) (\( u \) denotes the context). We may then write \( u[s] \) to denote a term obtained from \( u[t] \) by replacing \( t \) by \( s \). The symbol \( \bowtie (\equiv \text{or} \not\equiv) \) must denote identical symbols in the premise and in the conclusion of the rule. The rules are parameterized by the order \( \prec \) and by:

- A selection function \( \text{sel} \) that maps every clause \( C \) to a set of selected literals in \( C \). The set \( \text{sel}(C) \) must contain all \( \prec \)-maximal literals or at least one negative literal.
- A set of normalized constraints \( X \), closed under subset. Intuitively \( X \) denotes the set of abductible formulas, i.e., formulas that are allowed to be added as hypotheses.

The use of \( \prec \) and \( \text{sel} \) is standard, it aims at pruning the search space by restricting inferences, whereas the set \( X \) allows one to control the addition of hypotheses into the search space. The reader may consult [Nieuwenhuis and Rubio, 2001] for missing definitions and more details about the superposition calculus.

A crucial feature of the Superposition calculus is the availability of a general criterion for detecting redundant clauses. In \( SP \), a clause is considered redundant if all its instances are entailed by \( \bowtie \)-smaller instances of existing clauses. The definition for \( cSP \) is similar, with two differences: first the
constraints of the entailing clauses must be included in that of the considered clause, and second the literals occurring in this constraint can also be used in the entailment test, provided they are smaller than the considered clause. Formally:

**Definition 5** If $X$ is a constraint and $C$ is a clause, we denote by $X|_C$ the set of literals in $X$ that are smaller than or equal to $C$.

A $C$-clause $[C | X]$ is redundant w.r.t. a set of $c$-clauses $S$ if either $X$ is unsatisfiable or for every ground substitution $\sigma$ of the variables in $C$, there exist $c$-clauses $[D_i | Y_i] \in S$ ($1 \leq i \leq n$) and ground substitutions $\theta_i$ ($1 \leq i \leq n$) such that:

- $\forall i \in \{1 \ldots n\}, C\sigma \supseteq D_i\theta_i$ and $Y_i \subseteq X$, and
- $X|_{C\sigma}, D_1\theta_1, \ldots, D_n\theta_n \models C\sigma$.

A set of $c$-clauses is $S$- saturated if every $c$-clause deducible by the rules of $c$SP in one step is redundant w.r.t. $S$. A $c$SP-saturation of a set of $c$-clauses $S$ is a set of $c$-clauses $S^*$ such that: (i) every $c$-clause in $S$ is redundant w.r.t. $S^*$, (ii) every $c$-clause in $S^*$ is obtained from those in $S$ by a finite number of applications of the rules in $c$SP, (iii) $S^*$ is $c$SP-saturated.

The following theorem states the soundness and deductive completeness of $c$SP.

**Theorem 6** Let $S$ be a set of standard clauses and $S^*$ be a $c$SP-saturation of $S$. For every $X \in S$, $X^c$ is an $S$-implicate of $S$ iff $S^*$ contains a $c$-clause of the form $[\top | X^c]$ with $X^c \subseteq X^c$.

4. **On the Storage of Implicates**

Sets of implicates are huge, thus being able to store those sets in a compact way and to detect redundant implicates efficiently is a critical feature. We begin by devising an efficient algorithm for testing entailment between ground equational clauses.

**Definition 7** Let $C, D$ be two falsifiable ground clauses. The clause $D$ $\vDash$-subsumes $C$, written $D \leq_{\vDash} C$, iff the two following conditions hold:

1. for every negative literal $t \neq s \in D$, $t|_{C^c} = s|_{C^-}$;
2. for every positive literal $l \in D$, there exists a positive literal $m \in C$ such that $m|_{C^- \lor l^c} = t$ is a tautology.

**Theorem 8** Let $C$ and $D$ be two ground clauses. If $C$ and $D$ are falsifiable, then $D \models C$ iff $D \leq_{\vDash} C$.

Based on the entailment test of Definition 7, we define a trie-like data structure to store implicates and algorithms to remove redundancy inspired from [De Kleer, 1992]. A clausal tree is a tree whose edges are labeled by literals and whose leaves are labeled by either $\bot$ or $\top$. Each path from the root of the tree to a leaf can be associated with a clause defined as the disjunction of all the literals labeling the edges along the path. The set of clauses associated with the tree is the set of clauses associated with a path from the root to a $\top$-leaf. This representation ensures that the prefixes of the clauses will be shared, thus reducing the amount of consumed memory. Note that the edges pointing to a $\bot$-leaf are useless and may be deleted (a node with no successor may be labeled by $\bot$).

We consider a total ordering $<_\pi$ on ground literals, defined as follows: if $l_1$ is a negative literal and $l_2$ is a positive literal then $l_1 <_\pi l_2$, and if $l_1, l_2$ are literals with same polarity then $l_1 <_\pi l_2$ iff $l_1 < l_2$. To enforce maximal sharing and minimize the size of the tree, we assume that the literals occurring along a path in the tree are ordered w.r.t. $<_\pi$, i.e., for any path $e_1, \ldots, e_n$ in the tree, if $e_1, \ldots, e_n$ are literals labeled by literals $l_1, \ldots, l_n$ respectively, then $l_1 <_\pi \ldots <_\pi l_n$.

When a new implicate $C$ is generated, we need to test whether it is a consequence of a previously generated implicate (forward subsumption). If this is the case, then $C$ is redundant and can be dismissed, otherwise we need to delete in the clausal tree all branches corresponding to an implicate that is a logical consequence of $C$ (backward subsumption). We briefly sketch algorithms for performing these two tasks. The formal definitions of the algorithms can be found in [Echenim et al., 2017] together with detailed proofs of their properties.

**Forward Subsumption.** Beside the new generated clause $C$ and the clausal tree $T$, we also consider two input clauses $N$ and $M$ that are both initially empty. The clause $N$ contains the literals occurring in the parent nodes in the recursive calls and the clause $M$ permits to keep track of the negative literals of $C$ in recursive calls after having used them a first time to rewrite literals in the tree.

The algorithm is based on a depth-first traversal of the tree which is best described as a non-deterministic recursive algorithm. The algorithm returns true if the tree contains a single node labeled by $\top$ (representing the empty clause) and backtracks if the tree is not $\top$ and $C = \emptyset$ (base cases). Otherwise, an edge $e$ starting from the root of the tree is chosen. Let $l$ be the literal labeling $e$ and let $T'$ be the subtree pointed by $e$:

- If it is clear that $l \models M \lor C$, then we add $l$ into $N$ and we proceed to $T'$. Otherwise, the entailment condition is tested by checking that $l|_M$ is a contradiction (if $l$ is negative) or that $C|_{M \lor l^c}$ contains a tautological literal (if $l$ is positive).

- If the relation between $l$ and $M \lor C$ is not currently determined (because $l$ is $<_\pi$-greater than the literal currently considered in $C$ and thus may entail literals that remain to be examined), then the minimal literal of $C$ is added to $M$ before restarting the exploration of the branch corresponding to $e$.

- Lastly, if it is clear that $l \not\models C$, which is the case e.g. when $l$ is $<_\pi$-smaller than all the literals in $C$, then the algorithm backtracks.

**Backward Subsumption.** This handles the removal from a clausal tree $T$ of all the clauses $D$ that are $\vDash$-subsumed by a given c-clause $C$, under the assumption that $C$ itself is not subsumed by a clause stored in $T$. As the previous one, the algorithm performs a depth-first traversal of $T$ with a rewriting of literals. The main difference is that the roles of $C$ and $T$ are switched: the literals of a branch of $T$ are rewritten using the negative literals of $C$. When the subsumption test
succeeds, the algorithm cuts the corresponding branch in $T$ before exploring the remaining branches.

## 5 Experimental Results

Our prime implicate generation method has been implemented in a research prototype called cSP, written in OCaml, based on the LogTk library [Cruanes, 2014] for term ordering and congruence closure. We also implemented another version of the program, called cSP\_flat, that only handles flat clauses, i.e., clauses not containing function symbols of an arity strictly greater than 0, with additional refinements that only apply to this fragment.

We compared cSP and cSP\_flat with the most efficient available systems for generating implicates of logical formulas, namely primer [Previti et al., 2015] and Zres [Simon and Del Val, 2001], two propositional prime implicate generation tools based on satisfiability encoding and resolution respectively, and SOLAR [Iwanuma et al., 2009], a prime implicate generation tool for first order logic based on semantic tableaux. Equality is not built-in in these systems, thus the equality axioms had to be added into the considered formula: reflexivity, commutativity, transitivity and substitutivity. For Zres and primer, these axioms must also be grounded and transformed into propositional logic by flattening.

We compared the systems on two randomly generated sets of (flat and non-flat respectively) equational ground clauses of small size. Figure 2 compares the execution time of cSP on the random flat benchmark with that of cSP\_flat and primer. It shows that cSP\_flat is the most suitable tool to handle such problems.

<table>
<thead>
<tr>
<th>successes</th>
<th>SOLAR successes</th>
<th>Zres successes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cSP</td>
<td>76%</td>
<td>0% 0.042 99 21</td>
</tr>
<tr>
<td>primer</td>
<td>53%</td>
<td>4% 6.622 2275 74</td>
</tr>
<tr>
<td>cSP_flat</td>
<td>63%</td>
<td>9% 5.376 75 59</td>
</tr>
<tr>
<td>SOLAR</td>
<td>13%</td>
<td>10% 15.842 663190 606</td>
</tr>
<tr>
<td>Zres</td>
<td>52%</td>
<td>13% 0.695 X 2986</td>
</tr>
<tr>
<td>primer</td>
<td>53%</td>
<td>13% 0.794 X 2986</td>
</tr>
<tr>
<td>cSP_flat</td>
<td>63%</td>
<td>9% 5.376 75 59</td>
</tr>
<tr>
<td>cSP</td>
<td>76%</td>
<td>0% 0.042 99 21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>successes</th>
<th>cSP_flat successes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cSP</td>
<td>2%</td>
</tr>
<tr>
<td>primer</td>
<td>6%</td>
</tr>
<tr>
<td>cSP_flat</td>
<td>2%</td>
</tr>
<tr>
<td>SOLAR</td>
<td>59%</td>
</tr>
<tr>
<td>Zres</td>
<td>2%</td>
</tr>
<tr>
<td>primer</td>
<td>6%</td>
</tr>
<tr>
<td>cSP_flat</td>
<td>2%</td>
</tr>
<tr>
<td>cSP</td>
<td>2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>successes</th>
<th>timeouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>cSP</td>
<td>0%</td>
</tr>
<tr>
<td>Zres</td>
<td>33%</td>
</tr>
<tr>
<td>primer</td>
<td>31%</td>
</tr>
<tr>
<td>cSP_flat</td>
<td>19%</td>
</tr>
<tr>
<td>cSP</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 1: Test results summary: random non-flat benchmark

Table 1 compares Zres, primer and SOLAR with cSP\_flat and cSP on the random non-flat benchmark. For each system, we isolate the benchmarks for which the system is able to generate all implicates, and we specify for all systems the failure rate on those benchmarks (i.e., the percentage of formulas for which the system is not capable of generating all implicates in the allocated time, namely 5 min.), the time needed to generate all prime implicates in case of success, the number of inferences and the number of implicates generated. As shown in the ’successes’ column, cSP is the obvious winner in terms of the number of tests handled before timeout.

More detailed experimental results are presented in [Echenim et al., 2017].

## References

[Cruanes, 2014] Simon Cruanes. Logtk: a logic toolkit for automated reasoning and its implementation. In 4th Work-


