Quantifier-Free Equational Logic and Prime Implicate Generation

Mnacho Echenim[1,2], Nicolas Peltier[1,4] and Sophie Tourret[1,3]

[1] Grenoble Informatics Laboratory
[2] Grenoble INP - Ensimag
[3] Université Grenoble 1
[4] CNRS

Abstract. An algorithm for generating prime implicates of sets of equational ground clauses is presented. It consists in extending the standard Superposition Calculus with rules that allow attaching hypotheses to clauses to perform additional inferences. The hypotheses that lead to a refutation represent implicates of the original set of clauses. The set of prime implicates of a clausal set can thus be obtained by saturation of this set. Data structures and algorithms are also devised to represent sets of *constrained* clauses in an efficient and concise way. Our method is proven to be correct and complete. Practical experimentations show the relevance of our method in comparison to existing approaches for propositional or first-order logic.

1 Introduction

We tackle the problem of generating the prime implicates of a quantifier-free equational formula. From a formal point of view, an implicate of a formula S is a clause C such that $S \models C$, and this implicate is prime if for all implicates D such that $D \models C$, we have $C \models D$. In other words, prime implicates are the most general clausal consequences of a formula, and their generation is a more difficult problem than checking satisfiability. Prime implicate generation has many natural applications in artificial intelligence and system verification. It has been extensively investigated in the context of propositional logic [6,12,13,16,17,24,26], but there have been only very few approaches dealing with more expressive logics [14,15,18,19]. The approaches that are capable of handling first-order formulæ are based mainly on unrestricted versions of the resolution calculus (with an explicit encoding of equality axioms) or extensions of the tableau method and do not handle equality efficiently. More recently, algorithms were devised to generate sets of implicants of formulæ interpreted in decidable theories [8], by combining quantifier-elimination (for discarding useless variables) with model building (to construct sufficient conditions for satisfiability). The approach does not apply to equational formulæ with function symbols since this would involve second-order quantifier elimination.

In previous work [9,10] we devised procedures for generating implicates of equational formulæ containing only constant symbols. In this paper, we propose

an approach to handle arbitrary uninterpreted function symbols. There are two parts to our contribution. First, in Sect. 3, a calculus is devised to generate implicates. It is based on the standard rules of the Superposition Calculus [20], together with *Assertion* rules allowing the addition of new hypotheses during the proof search. These hypotheses are attached to the clauses as constraints, and once an empty clause is derived, the associated constraint corresponds to the negation of an implicate. This algorithm is completely different from that of [9]: its main advantage is that it remains complete even when applying all the usual restrictions of the Superposition Calculus, and that it allows for a better control of the generated implicates, in case the user is interested only to search for implicates of some particular form. Second, in Sect. 4 we extend the representation mechanism of [9] that uses a trie-based representation of equational clause sets in order to handle function symbols. This extension is not straightforward since, in contrast to [9], we have to encode substitutivity as well as transitivity. We devise data-structures and algorithms to efficiently store equational sets up to redundancy, taking into account the properties of the equality predicate. In Sect. 5 we experimentally compare our approach with existing tools [19,24] for propositional logic and first-order logic respectively. We also compare our method with the approach consisting in encoding equational formulæ as flat clauses by explicitly adding substitutivity axioms and applying the algorithms from [10]. Due to space limitations, the formal proofs are omitted. Proofs are all available at `http://membres-lig.imag.fr/tourret/documents/EPT15-long.pdf`.

2 Clauses with Uninterpreted Functions in Equational Logic

The theory of equational logic with uninterpreted functions will be denoted by EUF (see [2] for details). Let Σ be a *signature*, and Σ_n the function symbols in Σ of arity n, usually denoted by f, g (and a, b for Σ_0). The notation $\mathfrak{T}(\Sigma)$ stands for the set of well-formed ground terms over Σ, most often denoted by s, t, u, v, w. A well-founded reduction order $\prec$ on $\mathfrak{T}(\Sigma)$ such as Knuth-Bendix Ordering or Recursive Path Ordering [7] is assumed to be given. The subterm of t at position p is denoted by $t|_p$.

A *literal*, usually denoted by l or m, is either an equation (or *atom*, or *positive* literal) $s \simeq t$, or an inequation $s \not\simeq t$ (or *negative* literal). The literal written $s \bowtie t$ can denote either the equation or the inequation between s and t. The literal l^c stands for $s \not\simeq t$ (resp. $s \simeq t$) when l is $s \simeq t$ (resp. $s \not\simeq t$). A literal of the form $s \not\simeq s$ is called a *contradictory* literal (or a contradiction) and a literal of the form $s \simeq s$ is a *tautological* literal (or a tautology). We consider *clauses* as disjunctions (or multisets) of literals and *formulæ* as sets of clauses. If C is a clause and l a literal, $C \backslash l$ denotes the clause C where *all* occurrences of l have been removed (up to commutativity of equality). In Sect. 3, we also consider conjunctions of literals, called *constraints*. For every constraint $\mathcal{X} = \bigwedge_{i=1}^{n} l_i$, $\neg\mathcal{X}$ denotes the clause $\bigvee_{i=1}^{n} l_i^c$. Similarly, if $C = \bigvee_{i=1}^{n} l_i$ then $\neg C \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} l_i^c$. Empty

clauses and constraints are denoted by $\Box$ and $\top$ respectively. We often identify sets of clauses with conjunctions.

We define an *equational interpretation* $\mathcal{I}$ as a congruence relation on $\mathfrak{T}(\Sigma)$. A positive literal $l = s \simeq t$ is evaluated to $\top$ (true) in $\mathcal{I}$, written $\mathcal{I} \models l$, if $s =_{\mathcal{I}} t$; otherwise l is evaluated to $\bot$ (false). A negative literal $l = s \not\simeq t$ is evaluated to $\top$ in $\mathcal{I}$ if $s \neq_{\mathcal{I}} t$, and to $\bot$ otherwise. This evaluation is extended to clauses and sets of clauses in the usual way. An interpretation that evaluates C to $\top$ is a *model* of C (often written $\mathcal{M}$ in this paper). A *tautological clause* (or *tautology*) is a clause of which all equational interpretations are models and a *contradiction* is a clause that has no model.

We now associate every clause C with an equivalence relation $\equiv_C$ among terms, defined as the equality relation modulo the constraint $\neg C$, i.e., the smallest congruence containing all pairs (t, s) such that $t \not\simeq s \in C$.

Definition 1. *Let C be a clause, we define for any term s the C-*equivalence class *of s as $[s]_C = \{t \in \mathfrak{T}(\Sigma) \mid \neg C \models s \simeq t\}$. The corresponding equivalence relation is written $\equiv_C$. The C-*representative *of a term s, a literal l and a clause D are respectively defined by $s_{\downarrow C} \stackrel{def}{=} \min_{\prec}([s]_C)$, $l_{\downarrow C} \stackrel{def}{=} s_{\downarrow C} \bowtie t_{\downarrow C}$, for $l = s \bowtie t$, and $D_{\downarrow C} \stackrel{def}{=} \{l_{\downarrow C} \mid l \in D\}$*

Example 2. Let $a \prec b \prec c \prec d \prec e \prec g(b) \prec g(c)$ and $C = a \not\simeq g(c) \vee b \not\simeq c \vee d \simeq e$. The clause $D = g(b) \simeq e$ is such that $D_{\downarrow C} = a \simeq e$ because $[c]_C = \{b, c\}$, $[g(b)]_C = \{a, g(b), g(c)\}$ and $[e]_C = \{e\}$.

The two following orders on literals are used throughout the paper. Both orders are extended to clauses using the multiset extension and are relaxed (into $\preceq$ and $\leq_\pi$ resp.) by also accepting equal literals or clauses.

1. The total order $\prec$ on terms is extended to literals by considering that a negative literal $t \not\simeq s$ is a set $\{\{t, s\}\}$ and that a positive literal $s \simeq t$ is $\{\{t\}, \{s\}\}$ (see [22]).
2. The total order $<_\pi$ on literals is defined as follows:
   - the equations are all greater than the inequations;
   - for l_1 and l_2 literals with the same polarity, $l_1 <_\pi l_2$ iff $l_1 \prec l_2$.

The order $\prec$ is used, as is usual, to determine which implicates are prime and which are redundant (see Definition 14). The order $<_\pi$ is useful for handling clauses as presented in Sect. 4, but is not used outside of this scope.

Example 3. Let $C = g(a) \not\simeq b \vee c \simeq d$ and $D = a \not\simeq b \vee f(c) \simeq d$, with $a \prec b \prec c \prec d \prec f(c) \prec g(a)$. We have $D \prec C$ and $C <_\pi D$, because on the one hand $a \not\simeq b \prec c \simeq d \prec f(c) \simeq d \prec g(a) \not\simeq b$, and on the other hand $a \not\simeq b <_\pi g(a) \not\simeq b <_\pi c \simeq d <_\pi f(c) \simeq d$.

In propositional logic, testing entailment amounts to a simple inclusion test [6] but things are more complex in EUF because the axioms of transitivity and substitutivity must be taken into account. For example, the clause $e \not\simeq b \vee b \not\simeq c \vee f(a) \simeq f(b)$ is a logical consequence of the clause $e \not\simeq c \vee a \simeq c$ because of these axioms. The following theorem describes the so-called *projection* method for testing entailment in a syntactic way.

Theorem 4. *Let C and D be two non-tautological clauses. The relation $D \models C$ holds iff for every negative literal l in D, the literal $l_{\downarrow C}$ is a contradiction and for every positive literal l in D, there exists a positive literal m in C such that $m_{\downarrow C \vee l^c}$ is tautological.*

Example 5. Given the order $a \prec b \prec c \prec e \prec f(a) \prec f(b)$ on terms, let $D = e \not\simeq c \vee a \simeq c$ and $C = e \not\simeq b \vee b \not\simeq c \vee f(a) \simeq f(b)$, and let $l = e \not\simeq c$ and $m = a \simeq c$ be the literals of D. We have $l_{\downarrow C} = b \not\simeq b$ because $[b]_C = \{b, c, e\}$ and $\min_{\prec}([b]_C) = b$. Moreover the literal $f(a) \simeq f(b) \in C$ is such that $(f(a) \simeq f(b))_{\downarrow C \vee m^c} = f(a) \simeq f(a)$, hence C is redundant w.r.t. D.

In order to avoid having to handle large numbers of equivalent clauses, we define a clausal normal form that is unique up to equivalence.

Definition 6. *A non-tautological clause C is in* normal form *if:*

1. *every negative literal l in C is such that $l_{\downarrow C \setminus l} = l$;*
2. *every literal $t \simeq s \in C$ is such that $t = t_{\downarrow C}$ and $s = s_{\downarrow C}$;*
3. *there are no two distinct positive literals l, m in C such that $m_{\downarrow l^c \vee C}$ is a tautology;*
4. *C contains no literal of the form $t \not\simeq t$;*
5. *the literals in C occur exactly once in C.*

The normal form equivalent to C is denoted by $C_{\downarrow}$.

In our previous work on prime implicate generation [9], the focus was on strictly flat clauses (i.e., that contain only constant symbols). For the sake of handling non-flat clauses, the clausal normal form (see [10], Def. 4) had to be extended. The differences lie with points 1 and 3 of Def. 6. They respectively strengthen the requirements on negative and positive literals to cover the non-flat ones.

Example 7. Using the same term ordering as in Ex. 5, the clause $c \not\simeq b \vee e \not\simeq b \vee f(b) \simeq f(a)$ is the normal form of the clauses $c \not\simeq b \vee e \not\simeq b \vee f(c) \simeq f(a)$, $c \not\simeq b \vee e \not\simeq b \vee f(e) \simeq f(a)$, $c \not\simeq e \vee e \not\simeq b \vee f(b) \simeq f(a)$, etc.

Theorem 8. *The normal form of a non-tautological clause C is the $\prec$-smallest clause equivalent to C.*

3 Implicate Generation

Definition 9. *A* constrained clause *(or* c-clause*) is a pair $[C \,|\, \mathcal{X}]$ where C is a clause and $\mathcal{X}$ is a constraint.*

$[C \mid \top]$ is often written simply as C and referred to as a standard clause. A constraint is *normalized*, or in *normal form*, if the clause $\neg\mathcal{X}$ is in normal form. Note that only non-contradictory constraints can be normalized. Semantically, a constrained clause $[C \,|\, \mathcal{X}]$ is equivalent to the standard clause $\neg\mathcal{X} \vee C$. For example the c-clause $[c \simeq b \,|\, f(a) \simeq c \wedge c \not\simeq d]$ is equivalent to $c \simeq b \vee f(a) \not\simeq$

Superposition	$\dfrac{[r \simeq l \vee C \,|\, \mathcal{X}] \quad [u \bowtie v \vee D \,|\, \mathcal{Y}]}{[u[l] \bowtie v \vee C \vee D \,|\, \mathcal{X} \wedge \mathcal{Y}]}$	If $u\|_p = r$, $r \succ l$, $u \succ v$, and $(r \simeq l)$ and $(u \bowtie v)$ are selected in $(r \simeq l \vee C)$ and $(u \bowtie v \vee D)$ respectively.
Factoring	$\dfrac{[t \simeq u \vee t \simeq v \vee C \,|\, \mathcal{X}]}{[t \simeq v \vee u \not\simeq v \vee C \,|\, \mathcal{X}]}$	If $t \succ u$, $t \succ v$ and $(t \simeq u)$ is selected in $t \simeq u \vee t \simeq v \vee C$.

Table 1: Standard Inference Rules

$c \vee c \simeq d$. Intuitively, the intended meaning of a c-clause $[C \mid \mathcal{X}]$ is that the clause C can be inferred provided the literals in $\mathcal{X}$ are added as axioms to the considered clause set. The usual notion of redundancy is extended to c-clauses.

Definition 10. *A c-clause $[C \mid \mathcal{X}]$ is* redundant *w.r.t. a set of c-clauses S if either $\mathcal{X}$ is unsatisfiable or there exist c-clauses $[D_i \,|\, \mathcal{Y}_i] \in S$ $(1 \leq i \leq n)$ such that $\forall i \in \{1 \ldots n\}\, D_i \preceq C$ and $\mathcal{Y}_i \subseteq \mathcal{X}$, and $\mathcal{X}', D_1, \ldots, D_n \models C$, where $\mathcal{X}'$ denotes the set of literals in $\mathcal{X}$ that are $\prec$-smaller than C.*

We now present an extension of the standard superposition calculus [20] to a constrained superposition calculus referred to as c$\mathcal{SP}$, that is able to generate all prime implicates of a formula up to redundancy. This calculus is composed of the standard superposition rules extended to constrained clauses (Table 1) along with two assertion rules (Table 2). As usual the calculus is parameterized by the ordering $\succ$ on terms and by a selection function sel, where s$el(C)$ contains all maximal literals in C or at least one negative literal. A literal is *selected* in C if it occurs in s$el(C)$. We assume that the clausal part of c-clauses is systematically normalized, which explains the absence of the reflexion rule from Table 1. Note, however, that the constraint part is *not* normalized. Instead, the rules apply only if the constraint of the conclusion is already in normal form, up to the deletion of repeated literals. This strategy greatly prunes the search space, since many inferences can be dismissed. It also preserves deductive-completeness, since intuitively, one can always assume that implicates are in normal form.

Example 11. Consider the following c-clauses (with $f(a) \succ d \succ c \succ b \succ a$): $C : [f(a) \simeq b \mid d \not\simeq c]$, $D : [f(a) \simeq c \mid d \not\simeq c]$, $E : [f(a) \simeq c \mid d \not\simeq a]$. The Superposition rule applies on C and D, yielding: $[b \simeq c \,|\, d \not\simeq c]$ (the conjunction $d \not\simeq c \wedge d \not\simeq c$ is replaced by $d \not\simeq c$). However, the rule does not apply on C and E because the constraint $d \not\simeq c \wedge d \not\simeq a$ is not in normal form.

The principle of c$\mathcal{SP}$ is to generate the implicates of a formula as constraints of the empty clause. The standard inference rules are used to refute the clausal part of c-clauses, while the assertion rules explore the possible implicates by making hypotheses about their literals, and these are stored in the constraint part of the c-clauses. Since only the c-clauses with a refutable clausal part are of interest, the addition of new hypotheses is done only if these hypotheses render a new superposition inference possible (into the clause to which the rule applies for the Pos. Assert. rule and into the asserted literal for the Neg. Assert. rule).

Positive Assertion	$\dfrac{[u \bowtie v \vee C \,|\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,|\, \mathcal{X} \wedge t \simeq s]}$	If $u|_p = t$, $t \succ s$, $u \succ v$ and $(u \bowtie v)$ is selected in $(u \bowtie v \vee C)$.
Negative Assertion	$\dfrac{[t \simeq s \vee C \,|\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,|\, \mathcal{X} \wedge u \bowtie v]}$	If $u|_p = t$, $t \succ s$, $u \succ v$, and $(t \simeq s)$ is selected in $(t \simeq s \vee C)$.

Table 2: Assertion Rules

In other words, these rules use the fact that $S \models C$ iff $S \wedge \neg C \models \square$ to build implicates literal by literal.

Example 12. The following example shows how to derive the implicate $a \not\simeq d \vee f(c) \simeq f(b)$ from $\{a \simeq b, f(c) \simeq f(d)\}$, given the term ordering $a \prec b \prec c \prec d \prec f(a) \prec f(b) \prec f(c) \prec f(d)$.

$1\ [f(c) \simeq f(d) \,|\, \top]$	(hyp)
$2\ [f(c) \simeq f(a) \,|\, a \simeq d]$	(Pos. AR, 1)
$3\ [f(a) \not\simeq f(b) \,|\, a \simeq d \wedge f(c) \not\simeq f(b)]$	(Neg. AR, 2)
$4\ [a \simeq b \,|\, \top]$	(hyp)
$5\ [f(a) \not\simeq f(a) \,|\, a \simeq d \wedge f(c) \not\simeq f(b)]$	(Sup. 3, 4)
$6\ [\square \,|\, a \simeq d \wedge f(c) \not\simeq f(b)]$	(Ref. 5)

The negation of $a \simeq d \wedge f(c) \not\simeq f(b)$ is the desired implicate. Note for instance that the addition of the hypothesis $a \simeq d$ in Clause 1 was possible because it allowed one to replace constant d by a. The Assertion rules merge in a single rule the addition of a new hypothesis followed by a superposition inference from or into this hypothesis.

Theorem 13. *c$\mathcal{SP}$ is sound and deductive-complete, i.e., for any set of clauses S, C is a non-tautological implicate of S iff c$\mathcal{SP}$ generates from S a c-clause $[\square \,|\, \mathcal{X}]$ such that $\neg\mathcal{X} \models C$.*

Note that S possibly admits infinitely many prime implicates (e.g., $a \simeq b, c \simeq d \models f(a, c, t) \simeq (b, d, t)$ for every term t). Furthermore, S is not necessarily equivalent to its set of prime implicates, for instance $\{f(a) \simeq a, f(b) \simeq b, a \not\simeq b\} \models f^n(a) \not\simeq f^n(b)$, for every $n \in \mathbb{N}$ and none of the $f^n(a) \not\simeq f^n(b)$ is prime, because $f^{n+1}(a) \not\simeq f^{n+1}(b) \models f^n(a) \not\simeq f^n(b)$ holds for any $n \in \mathbb{N}$.

4 Clause Storage and Redundancy Detection

To store the clauses generated by c$\mathcal{SP}$ and efficiently detect redundancies, a trie-like data structure, the *clausal tree*, is used. It allows one to store efficiently and concisely sets of clauses while taking into account equality axioms. Note that, since our goal is to generate all prime implicates of a formula, we only

test one-to-one entailment between clauses (in contrast to the usual practice in automated deduction we cannot discard clauses that are redundant w.r.t. more than one clause since this clause may well be prime). For this reason, we define *e-subsumption*, and we assimilate it to redundancy in the rest of this article.

Definition 14. *Let C and D be two clauses. The clause C* e-subsumes *the clause D, written $C \leq_e D$, iff $C \models D$ and $C \preceq D$. A c-clause $[C \mid \mathcal{X}]$* c-subsumes *a clause $[D \mid \mathcal{Y}]$, written $[C \mid \mathcal{X}] \leq_e [D \mid \mathcal{Y}]$) iff $C \leq_e D$ and $\mathcal{X} \subseteq \mathcal{Y}$.*

Note that both parts of the c-clauses are handled in different ways: the inclusion relation $\subseteq$ used to compare constraints is clearly stronger than the e-subsumption relation $\leq_e$ used for clauses. For instance we have (if $a \succ b \succ c$):

$$\begin{array}{lll} [a \not\simeq b \vee f(b) \simeq f(d) \mid \top] & \leq_e & [a \not\simeq c \vee b \not\simeq c \vee f(c) \simeq f(d) \mid \top], \text{ but} \\ [\Box \mid a \simeq b \wedge f(b) \not\simeq f(d)] & \not\leq_e & [\Box \mid a \simeq c \wedge b \simeq c \wedge f(c) \not\simeq f(d)]. \end{array}$$

Clausal trees are similar to the tries of propositional logic that are trees where the edges are labeled with literals and where some additional ordering constraints ensure the efficiency of the search algorithms. In such a tree, the represented clauses are the branches, that is the disjunction of the literals labeling the edges from root to leaf.

Definition 15. *A* clausal tree *is inductively defined as either $\Box$, or a set of pairs of the form (l, T') where l is a literal and T' a clausal tree. In addition, a clausal tree T with $(l, T') \in T$ must respect the following conditions:*

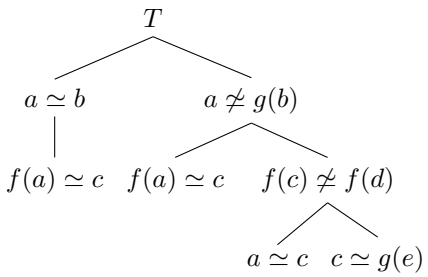
- *for all l' appearing in T', $l' <_\pi l$,*
- *there is no clausal tree $T'' \neq T'$ such that $(l, T'') \in T$.*

The set of clauses represented by a clausal tree T is defined inductively as follows:

$$\mathcal{C}(T) = \begin{cases} \{\Box\} & \text{if } T = \Box \\ \displaystyle\bigcup_{(l,T') \in T} \left(\bigcup_{D \in \mathcal{C}(T')} \{l \vee D\} \right) & \text{otherwise.} \end{cases}$$

As the definition implies, leaves can be either $\Box$ or $\emptyset$, but in practice if a leaf is labeled with $\emptyset$ (a failure node) then the corresponding branch is irrelevant because a tree of the form $T \cup \{(l, \emptyset)\}$ can be replaced by T without affecting the represented set. The only exception is the empty tree, in which the root is labeled with $\emptyset$. A clausal tree is *normalized* if all the clauses in $\mathcal{C}(T)$ are in normal form. In the following, we assume that all clausal trees are normalized.

Example 16. The structure T below is a clausal tree with the term order $a \prec b \prec c \prec g(c) \prec g(e) \prec f(c) \prec f(d)$. No leaf is labeled with $\emptyset$, and for a better readability the labels are associated with the nodes rather than with the edges leading to them.



The clauses in $\mathcal{C}(T)$ are:

$a \simeq b \vee f(a) \simeq c$
$a \not\simeq g(b) \vee f(a) \simeq c$
$a \not\simeq g(b) \vee f(c) \not\simeq f(d) \vee a \simeq c$
$a \not\simeq g(b) \vee f(c) \not\simeq f(d) \vee c \simeq g(e)$

Notation 17 *Let C be a clause in normal form and T be a clausal tree such that $\forall D \in \mathcal{C}(T)$, $C \vee D$ is in normal form and $\forall l \in D$, $C <_\pi l$. In this case, $C.T$ denotes the clausal tree T' such that $\mathcal{C}(T') = \{C \vee D \mid D \in \mathcal{C}(T)\}$.*

The storage of constrained clauses is similar to that of standard clauses. A main clausal tree is used to store the clausal part of constrained clauses and at each leaf of this tree, a trie is appended to store the different constraints associated to the same clause. Note that, according to Def. 10, constraints are compared using set inclusion instead of logical entailment [5], thus the second tree must be a trie and *not* a clausal tree. In addition, all generated implicates (c-clauses with an empty clausal part) should be stored in a clausal tree in order to remove non-prime implicates.

There are three main operations on clausal trees. The first one consists in checking whether a new clause is redundant w.r.t. an existing one already stored in a clausal tree. The second one removes from a clausal tree all clauses that are redundant w.r.t. a given clause. The last one is the insertion of a new clause into a clausal tree. This last operation is straightforward and thus is not described here. On the contrary, the first two operations are not trivial and are thus carefully detailed in the remaining parts of this section. The algorithms for c-clauses are neither theoretically nor technically challenging compared to the ones for standard clauses. Thus the choice was made to present them only for standard clauses.

The algorithm isEntailed (Alg. 1) tests whether a clause C is redundant w.r.t. a clause in $\mathcal{C}(T)$, where T is a clausal tree. To do so, a call is made to isEntailed$(T, \square, C, \square)$ and in the recursive calls to isEntailed(T', M, C', N), $M \vee C'$ is equal to C and N represents the path from the root of T to the subtree T'. The principle underlying these calls is to go through the input clause C and tree T while performing the operations necessary to test entailment with the projection method (Th. 4). Note that it is here that the use of the order $<_\pi$ is crucial. Intuitively, the need for this order stems from the fact that the negative literals of a clause C are the ones used to project C onto other clauses. In particular for the projection of positive literals, it is necessary to know of all the negative literals that belong to C, while the reverse does not hold.

5. Using logical entailment makes the calculus incomplete due to the deletion of clauses whose constraint is not in normal form.

Algorithm 1 isEntailed$(T,\ M,\ C,\ N)$

Require: T is a clausal tree in normal form, $M \vee C$ and N are clauses in normal form, M is negative and $N \models M \vee C$

Ensure: isEntailed$(T,\ M,\ C,\ N) = \top$ iff $\exists D \in \mathcal{C}(T), D \vee N \leq_e M \vee C$

1: **if** $T = \square$ **then return** $N \preceq M \vee C$
2: $T_1 \leftarrow \{(l, T') \in T \mid l_{\restriction M}$ is a contradiction$\}$
3: **if** $\bigvee_{(l,T') \in T_1}$ isEntailed$(T',\ M,\ C,\ N \vee l)$ **then return** $\top$
4: **if** $C = \square$ **then return** $\bot$
5: $m_1 \leftarrow \min_{<_\pi} \{m \in C\}$
6: **if** m_1 is of the form $u \not\approx v$, with $u \succ v$ **then**
7: $\quad T_2 \leftarrow \{(l, T') \in T \mid l_{\restriction M} \not<_\pi m_1$ and $\nexists w, (l_{\restriction M} = u \not\approx w$, with $u \succ w)\}$
8: $\quad$ **return** $\bigvee_{(l,T') \in T_2}$ isEntailed$(l.T',\ M \vee m_1,\ C \setminus m_1,\ N)$
9: **else**
10: $\quad T_3 \leftarrow \{(l, T') \in T \mid C_{\restriction M \vee l^c}$ contains a tautological literal$\}$
11: $\quad$ **return** $\bigvee_{(l,T') \in T_3}$ isEntailed$(T',\ M,\ C,\ N \vee l)$

Theorem 18. *If T is a clausal tree in normal form, $M \vee C$ and N are clauses in normal form, M is negative and $N \models M \vee C$ then the call* isEntailed(T, M, C, N) *terminates and* isEntailed$(T, M, C, N) = \top$ *iff* $\exists D \in \mathcal{C}(T), D \vee N \leq_e M \vee C$.

The algorithm pruneEntailed (Alg. 2) removes from the input tree T all the clauses redundant w.r.t. the input clause C. It proceeds by going through both objects, performing projections and storing the already considered literals in parameters N and M. Once an entailment is established in this way, all that remains is to compare the selected clauses using the order $\prec$ to detect redundancies. This last part is done by the algorithm pruneInf (Alg. 3).

Proposition 19. *Let C and N be clauses in normal form and T be a clausal tree in normal form verifying the preconditions of* pruneInf. *The output tree* $T_{out} =$ pruneInf(T, C, N) *is such that* $\mathcal{C}(T_{out}) = \{D_T \in \mathcal{C}(T) \mid C \not\leq_e D_T \vee N\}$.

Theorem 20. *Let $C \vee M$ and N be clauses in normal form and T be a clausal tree in normal form verifying the preconditions of* pruneEntailed. *Then the calls* pruneEntailed(T, M, C, N) *and* pruneInf(T, C, N) *always terminate and* $T_{out} =$ pruneEntailed(T, M, C, N) *is such that* $\mathcal{C}(T_{out}) = \{D \in \mathcal{C}(T) \mid C \vee M \not\leq_e D \vee N\}$.

Remark 21. The main difference with the flat version of the algorithms [9] is the handling of clausal tree branches labeled with positive literals, which had to be adapted to the redefined projection method. Also the case in which no redundancy is detected (resp. lines 4 & 5 of Alg. 1 & 2) has to be postponed. To ensure the correctness of the algorithms some recursive cases must now be checked first.

Algorithm 2 PRUNEENTAILED$(T,\ M,\ C,\ N)$

Require: T is a clausal-tree in normal form, $M \vee C$ and N are clauses in normal form, $M \models N$ and ISENTAILED$(N.T,\ \Box,\ C \vee M,\ \Box) = \bot$.
Ensure: $\mathcal{C}(T_{out}) = \{D \in \mathcal{C}(T) \mid C \vee M \not\leq_e D \vee N\}$, with $T_{out} =$ PRUNEENTAILED$(T,\ M,\ C,\ N)$.

if $C = \Box$ **then return** PRUNEINF$(T,\ M,\ N)$
select $m_1 \in C$ such that $m_{1\restriction N} = \min_{<_\pi} \{m_{\restriction N} \mid m \in C\}$
if $m_{1\restriction N}$ is a contradiction **then**
    return PRUNEENTAILED$(T,\ M \vee m_1,\ C \setminus m_1,\ N)$
if $T = \Box$ **then return** T
$T_1 \leftarrow \{(l, T') \in T \mid l = u \not\simeq v \wedge m_{1\restriction N} \succeq l\}$
$T_{out1} \leftarrow \{(l,$ PRUNEENTAILED$(T',\ M,\ C,\ N \vee l) \mid (l, T') \in T_1 \wedge$ PRUNEENTAILED$(T',\ M,\ C,\ N \vee l) \neq \emptyset\}$
if m_1 is positive **then**
    $T_2 \leftarrow T \setminus T_1$
    $T_{out2} \leftarrow \{(l,$ PRUNEENTAILED$(T',\ M \vee L_l,\ C \setminus L_l,\ N \vee l)) \mid (l, T') \in T_2 \wedge L_l = \{m \in C \mid l_{\restriction N \vee m}$ is tautological$\} \wedge$ PRUNEENTAILED$(T',\ M \vee L_l,\ C \setminus L_l,\ N \vee l) \neq \emptyset\}$
    return $T_{out1} \cup T_{out2}$
else
    return $T_{out1} \cup T \setminus T_1$

Algorithm 3 PRUNEINF(T, C, N)

Require: T is a clausal-tree in normal form, C in a clause in normal form, N is a clause in normal form, $C \models N$.
Ensure: $\mathcal{C}(T_{out}) = \{D \in \mathcal{C}(T) | C \not\leq_e D \vee N\}$, with $T_{out} =$ PRUNEINF(T, C, N).

if $T = \Box$ and $C \not\simeq N$ **then return** T
if $C \not\simeq N$ **then**
    return $\{(l,$ PRUNEINF$(T', C, N \vee l)) | (l, T') \in T \wedge$ PRUNEINF$(T', C, N \vee l) \neq \emptyset\}$
return $\emptyset$

5 Experimental Results

We have developed a prototype for generating EUF prime implicates. It uses the Logtk library [5] at its core for term manipulation, and for parsing TPTP inputs [25]. The c$\mathcal{SP}$ rules and the clausal tree operations are built into a Given-Clause loop [23] (in the *Otter* variant). To ensure termination, it is necessary to impose additional conditions on the generated implicates. Indeed, the set of implicates is infinite in general, e.g, $a \not\simeq b, f(a) \simeq a, f(b) \simeq b \models f^n(a) \not\simeq f^n(b)$. In the experiments we only computed implicates built on the set of ground terms occurring in the initial formula. We tested the tool on two sets of problems: a collection of randomly generated formulæ of small size and a set of benchmarks from the SMT-LIB library [3]. All the tests were conducted on a machine equipped with an Intel core i5-3470 CPU and 4×2 GB of RAM.

	successes	SOLAR successes			Zres successes			(flat-)cSP			timeouts	
		time(s)	inf.	PIs	time(s)	inf.	PIs	time(s)	inf.	PIs	inf.	PIs?
SOLAR	15%	11.842	663190	506	-	-	-	-	-	-	2452908	28152
Zres	52%	0.695	X	2986	12.474	X	13804	-	-	-	X	X
flat-cSP	63%	6.622	5157	74	2.334	3300	158	14.290	11005	348	68959	X
cSP	76%	0.042	110	21	3.436	1322	47	10.193	1834	79	14714	538

Table 3: Randomly generated formulæ - test results summary

The first experiment presented is a comparison of different prime implicate generation systems on a set of randomly generated formulæ with a timeout of 5 minutes. The selected systems are:

- Zres [6], a prime implicate generation tool for propositional logic [24],
- SOLAR [7], a prime implicate generation tool for first-order logic [19] which can handle equational formulæ through the use of modification methods [11],
- cSP, the prime implicate generator prototype for ground equational logic described in this paper,
- and flat-cSP, the former version of the cSP prototype, that only handles flat clauses.

To the best of our knowledge only two other prime implicate generation tools are currently available. One is ritrie [17], a tool that generates propositional prime implicates. This tool was outperformed by Zres in past experiments and we chose not to include it in this set of experiments. The second is the Mistral SMT solver [8] that cannot be compared with the other tools because its prime implicate generation is not complete. More generally the approach in [8] applies to any theory admitting quantifier-elimination but this property does not hold for the logic we consider in the present paper since the elimination of function symbols would require to handle second-order quantification. The input problems were flattened (see e.g. [4] for a definition) for flat-cSP and Zres, and the substitutivity axiom instantiated when necessary. Furthermore, for Zres, these flat equational problems were also converted to propositional ones, by instantiating the transitivity of equality when necessary. In order to perform meaningful comparisons, SOLAR has been parameterized to generate only implicates built on the considered ground terms. Note that Zres generates propositional implicates which can always be translated back into equational clauses built on these terms.

The test set consists of randomly generated formulæ of 2 to 4 clauses containing 1 to 3 literals each, with terms of depth between 0 and 2, based on signatures of either 3 or 6 symbols of arity 0 or 1 and 2 constants. Six formulæ are generated in each case, for a total of 144 benchmarks. Although the resulting formulæ are rather small, some of them are complex enough that they timeout on all systems and some produce tens of thousands of implicates, generated after millions of inferences.

The results are summarized in Table 3. Each line corresponds to a system. The column labeled 'successes' indicates the percentage of tests that were completed before the 5 minute timeout. The three columns under the label 'SOLAR

6. Many thanks to Prof. L. Simon for providing the executable file.
7. Many thanks to Prof. H. Nabeshima for providing the executable file.

successes' summarize average results on those tests on which SOLAR terminated before the timeout. The other columns contain results on tests on which Zres terminated but not SOLAR, and on which flat-cSP terminated but not Zres and SOLAR. Finally, the 'timeout' columns expose the mean results on all interrupted tests. Columns labeled 'time', 'inf.' and 'PIs' respectively give the mean execution time, mean number of inferences and mean number of prime implicates found for each set of tests. The last column is labeled 'PIs?' because due to the timeout, the implicates found are not guaranteed to be prime. Cells labeled with an 'X' indicate that the corresponding data is not accessible.

As shown in the 'successes' column, cSP is the obvious winner in terms of the number of tests handled before timeout. It should also be mentioned that cSP solves all the problems that other systems solve, except for two that are solved only by Zres. The 15% of problems solved by SOLAR are the simplest of the random formulæ. The results show that SOLAR's approach is very costly both in terms of time and space, although methods to reduce these costs are being investigated [8]. The high number of prime implicates this tool generates compared to those produced by cSP may seem surprising. In fact, SOLAR returns an over-approximation of the result because it does not take into account the equality axioms in its redundancy detection. Thus for example, any literal $t \simeq s$ also appears as $s \simeq t$ and $f(s) \simeq f(t)$ is not detected as redundant w.r.t. $s \simeq t$. Comparatively, the huge number of prime implicates generated by Zres is not surprising at all. It stems directly from the propositional translation of the initial problems and the introduction of new propositional variables. Although Zres is faster than cSP on the problems they both solve, it solves only 52% of the problems, while cSP solves 76% of them. The results in the '(flat-)cSP' column are globally higher than those in the 'Zres successes' columns, because the most difficult benchmarks are solved only by cSP and to a lesser extend by flat-cSP. Since cSP solves more problems than flat-cSP and does so faster and with fewer clauses processed, cSP is clearly better adapted to dealing with originally non-flat formulæ. The number of inferences and generated non-redundant implicates when the tool times out illustrate the heavy cost of the cSP inferences and redundancy detection mechanism compared to that of SOLAR. It is a price that seems partly unavoidable to eliminate all redundancies, since this requires complex algorithms.

The second experiment presented uses benchmarks from the QF_AX logic of SMT-LIB [3]. They are synthetic benchmarks that model some properties in the SMT theory of arrays with extensionality, namely:

- some swappings of elements between cells of an array are commutative (swap benchmarks),
- swapping elements between identical cells of equal arrays generate equal arrays (storeinv benchmarks)

The benchmarks labeled with invalid have been tweaked to falsify the property.

Given that c$\mathcal{SP}$ cannot handle smt-lib inputs or the theory of arrays, we preprocessed the benchmarks by first converting them to TPTP using the

8. Personal communication of Prof. Nabeshima

SMTtoTPTP tool[9] and then applying the method described in [4] to generate equisatisfiable problems free of the axioms of the theory of arrays with extensionality. As shown in [1], these problems can be nontrivial to solve even for state-of-the-art theorem provers like E [23] and one cannot expect that the entire set of prime implicates can be generated in reasonable time. We use them mainly to evaluate the impact of our redundancy-pruning technique on the number of superposition inferences carried out by blocking the Assertion rules inferences, allowing the comparison of cSP with the E theorem prover. The main differences between the methods are the normalization of clauses and the redundancy pruning mechanism. On the one hand, the redundancy pruning algorithm used by cSP is weaker because it does not allow for equational simplification or other n-to-one redundancy pruning rules. On the other hand one-to-one redundancy testing is stronger since its uses logical entailment instead of subsumption. The comparison of cSP with the E theorem prover on these benchmarks shows that the normalization approach can, in some nontrivial cases, reduce the number of processed clauses by an order of magnitude.

Figure 1 presents the most notable results of this experiment, that is the results of the swap benchmarks. Among these, only the benchmarks on which both E and cSP (without Assertion rules) terminate before timeout and without memory overflow were kept, i.e. 76 out of 146. Squares represent the invalid benchmarks, i.e. the satisfiable formulæ, while crosses mark the unsatisfiable ones. An interesting observation is that for the largest invalid benchmarks, cSP needs to process a smaller number of clauses than E before terminating, even 10 times less in the case of the invalid_swap benchmarks. The unsatisfiable swap benchmarks were run with a timeout of 10 minutes (the triangles in Fig. 1) and the corresponding results hint that this phenomenon could also be true for larger unsatisfiable problems. This suggests that the redundancy pruning technique based on normalization and clausal trees could be profitably integrated into state-of-the-art superposition-based theorem-provers, at least for ground equational clause sets. However, it might not always be useful, for example the 10 out of 19 storeinv benchmarks that do not fail show the opposite tendency.

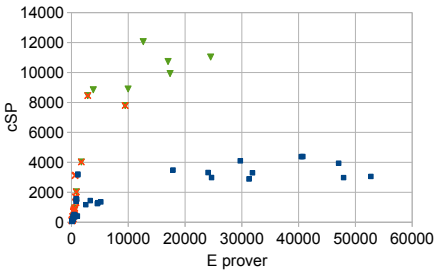


Fig. 1: swap benchmarks - comparison of the number of processed clauses for E and cSP.

6 Conclusion

In this article, a novel approach for the generation of prime implicates in ground equational logic is presented. It is proved sound and complete and ex-

9. http://users.cecs.anu.edu.au/~baumgart/systems/smttotptp/

periments are conducted to compare the approach to state-of-the-art tools. These show that `cSP` outperforms all other prime implicate generation systems on simple formulæ and can even tackle more involved problems than others, although none of the methods scale well. We also evaluate the impact of the normalization and pruning techniques of `cSP` compared to the redundancy detection of the `E` theorem prover. A potential improvement of redundancy detection using these techniques is highlighted. From a practical point of view, the implementation of the `cSP` prototype leaves rooms for many improvements, for instance a better selection strategy could be used. Note also that tries or clausal trees can be represented as directed acyclic graphs (where identical subtrees are shared) in order to merge suffixes as well as prefixes of clauses. A more drastic evolution would be to integrate `cSP` to an existing theorem prover to take advantage of its built-in optimizations. On the theoretical side, the c$\mathcal{SP}$ calculus can easily be extended, at least to handle variables, but the extensibility of the redundancy detection method has not been investigated yet. Well-known theoretical limitations may threaten such an extension since the entailment relation in full first-order logic is not decidable [21]. The frontier of what can and cannot be done on the generation of prime implicates in first-order logic is not yet clear and needs further investigations.

References

1. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans Comput Log.*, 10(1):1–51, 2009.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at ttwww.SMT-LIB.org.
4. M. P. Bonacina and M. Echenim. Theory decision by decomposition. *J. Symb. Comput.*, 45(2):229–260, 2010.
5. S. Cruanes. Logtk: A logic ToolKit for automated reasoning and its implementation. In *4th Workshop on Practical Aspects of Automated Reasoning.*, 2014.
6. J. De Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the National Conference on Artificial Intelligence*, pages 780–780. John Wiley & Sons ltd, 1992.
7. N. Dershowitz. Orderings for term-rewriting systems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 123–131, Washington, DC, USA, 1979. IEEE Computer Society.
8. I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken. Minimum satisfying assignments for SMT. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification*, number 7358 in Lecture Notes in Computer Science, pages 394–409. Springer, 2012.
9. M. Echenim, N. Peltier, and S. Tourret. An approach to abductive reasoning in equational logic. In F. Rossi, editor, *IJCAI 2013 - International Joint Conference on Artificial Intelligence*, pages 531–537, Beijing, China, Aug. 2013. AAAI Press.

10. M. Echenim, N. Peltier, and S. Tourret. A rewriting strategy to generate prime implicates in equational logic. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Automated Reasoning*, number 8562 in Lecture Notes in Computer Science, pages 137–151. Springer International Publishing, July 2014.
11. K. Iwanuma, H. Nabeshima, and K. Inoue. Toward an efficient equality computation in connection tableaux: A modification method without symmetry transformation—a preliminary report—. *First-Order Theorem Proving*, page 19, 2009.
12. P. Jackson. Computing prime implicates incrementally. *Autom. Deduc. CADE-11*, pages 253–267, 1992.
13. A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *J. Symb. Comput.*, 9(2):185–206, 1990.
14. E. Knill, P. T. Cox, and T. Pietrzykowski. Equality and abductive residua for horn clauses. *Theoretical Computer Science*, 120(1):1–44, Nov. 1993.
15. P. Marquis. Extending abduction from propositional to first-order logic. In P. Jorrand and J. Kelemen, editors, *Fundamentals of Artificial Intelligence Research*, number 535 in Lecture Notes in Computer Science, pages 141–155. Springer, 1991.
16. A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Autom. Reason. Anal. Tableaux Relat. Methods*, pages 250–264, 2009.
17. A. Matusiewicz, N. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. *Found. Intell. Syst.*, pages 203–213, 2011.
18. M. C. Mayer and F. Pirri. First order abduction via tableau and sequent calculi. *Log. J. IGPL*, 1(1):99–117, 1993.
19. H. Nabeshima, K. Iwanuma, K. Inoue, and O. Ray. SOLAR: An automated deduction system for consequence finding. *AI Commun.*, 23(2):183–203, Jan. 2010.
20. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, pages 371–443. 2001.
21. M. Schmidt-Schauss. Implication of clauses is undecidable. *Theor. Comput. Sci.*, 59:287–296, 1988.
22. S. Schulz. E - a brainiac theorem prover. *AI Commun*, 15(2-3):111–126, 2002.
23. S. Schulz. System description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc.of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.
24. L. Simon and A. Del Val. Efficient consequence finding. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 359–370. Lawrence Erlbaum Associates ltd, 2001.
25. G. Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *J. Autom. Reason.*, 43(4):337–362, 2009.
26. P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *IEEE Trans. Electron. Comput.*, EC-16(4):446–456, 1967.