An Approach to Abductive Reasoning in Equational Logic

Mnacho Echenim, Nicolas Peltier, Sophie Tourret

Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

An Approach to Abductive Reasoning in Equational Logic*

M. Echenim, N. Peltier, S. Tourret
University of Grenoble (CNRS, Grenoble INP/LIG)

Abstract

Abduction has been extensively studied in propositional logic because of its many applications in artificial intelligence. However, its intrinsic complexity has been a limitation to the implementation of abductive reasoning tools in more expressive logics. We have devised such a tool in ground flat equational logic, in which literals are equations or disequations between constants. Our tool is based on the computation of prime implicates. It uses a relaxed paramodulation calculus, designed to generate all prime implicates of a formula, together with a carefully defined data structure storing the implicates and able to efficiently detect, and remove, redundancies. In addition to a detailed description of this method, we present an analysis of some experimental results.

1 Introduction

Abductive reasoning (see for instance [Peirce, 1955]) is the process of inferring relevant hypotheses from data (as opposed to deduction, which consists in deriving logical consequences of axioms). Given a logical formula C, the goal is to compute a formula H such that the implication $H \Rightarrow C$ holds. This mode of reasoning can be used, e.g., to infer plausible explanations of observed facts. There exists an extensive amount of research on abductive reasoning, mainly in propositional logic, with numerous applications for instance in planning [Shanahan, 1989] or truth-maintenance in knowledge bases [De Kleer and Reiter, 1987]. Abduction can be performed in a top-down manner, by allowing some hypotheses to be asserted instead of being proven. However it is more often reduced to a consequence-generation problem: indeed, by contrapositive, the implication $H \Rightarrow C$ holds iff $\neg H$ is a logical consequence of $\neg C$. Thus explanations of C can be generated from the derivation of the logical consequences (i.e., the *implicates*) of the negation of C. In general, these explanations are further restricted to ensure relevance: for instance only explanations defined on a particular set of symbols, called the *abducible symbols* are considered. It is clear that the problem of generating all the implicates of a given formula is much more difficult than merely testing whether the latter is satisfiable. Existing proof procedures are tailored to *test* that a given formula is a logical consequence of a set of axioms (usually by *reductio ad absurdum*), and therefore are not well-adapted to *generate* all such implicates. Existing approaches for computing implicates are mostly restricted to propositional logic, where they are used by AI devices such as expert systems to minimize boolean functions. These approaches use either variants of the resolution rule (see, e.g., [Leitsch, 1997]), together with specific redundancy criteria and strategies ensuring efficiency [Tison, 1967; Kean and Tsiknis, 1990; Jackson, 1992; De Kleer, 1992], or decomposition-based approaches in the spirit of the DPLL method, which compute implicates by recursively decomposing them into smaller pieces [Rymon, 1994; Ramesh *et al.*, 1997; Matusiewicz *et al.*, 2009]. To the best of our knowledge, the only published papers in which the problem of abductive reasoning in more expressive logics has been considered are [Mayer and Pirri, 1993; Marquis, 1991; Mayer and Pirri, 1994]. In [Marquis, 1991], implicates are generated by using the resolution rule. This approach extends straightforwardly to first-order logic (using unification) and some specific classes for which termination can be ensured are defined, relying on well-known termination results for the resolution calculus, see for instance [Fermüller *et al.*, 2001]. In [Mayer and Pirri, 1993] a tableaux-based proof procedure is described for abductive reasoning. The principle is to apply the usual decomposition rules of propositional logic, and then to compute the formulæ that force the closure of all open branches in the tableaux, thus yielding sufficient conditions ensuring unsatisfiability. The approach is extended to first-order logic by using reverse skolemization techniques in order to eliminate the Skolem symbols introduced for handling existential quantifiers. This procedure has been extended to some modal logics [Mayer and Pirri, 1994]. As far as we are aware, there is no published work on abductive reasoning for equational formulæ.

In [Echenim and Peltier, 2012], we have proposed a method to extract ground abducible implicates of first-order formulæ, motivated by some applications in program verification. The method works by using a specifically tailored superposition-based calculus [Nieuwenhuis and Rubio, 2001] which is capable of generating, from a given set of first-order

*Work partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

clauses S with equality, a set of *ground* (i.e., with no variables) and *flat* (i.e., with no function symbols) clauses S' such that all abducible implicates of S are implicates of S'. If the formula at hand is satisfiable, these implicates can be seen as missing hypotheses explaining the "bad behavior" of the program (if the formula is unsatisfiable then the program is of course error-free). However, the proposed calculus is not able to generate *explicitly* the implicates of S'. This task is performed by a post-processing step which consists in translating the clause set S' into a propositional formula by adding relevant instances of the equality axioms, and then using the unrestricted resolution calculus to generate the propositional implicates. This approach is sound, complete and terminating, but it is also very inefficient, in particular due to the fact that a given clause may have several (in general, exponentially many) representants, that are all equivalent modulo the usual properties of the equality predicate. Computing and storing such a huge set of clauses is time-consuming and of no practical use.

The present paper addresses this issue. We devise a new algorithm for generating prime implicates of quantifier-free equational formulæ with no function symbols. It uses a more direct approach, in which the properties of the equality predicate are "built-in" instead of being explicitly encoded as axioms. This affects both the representation of the clauses, i.e., the way they are stored in the database and tested for redundancy, and their generation: instead of using the resolution method, new rules are devised, which can be viewed as a form of relaxed paramodulation. Our algorithm is proven to be sound, terminating and complete (i.e., it generates all implicates in a finite time, up to redundancy).

The paper is structured as follows. In Section 2, we briefly recall the basic definitions that are necessary for the understanding of our work. In Section 3, a new data-structure is introduced to allow for a compact storage of the clauses (up to equivalence) and algorithms are devised for storing and retrieving clauses. In Section 4, inference rules are presented to generate implicates in equational logic. Section 5 reports some experiments showing evidence of the practical interest of our approach (w.r.t. the translation-based approach, using state-of-the-art systems for propositional logic). Section 6 briefly concludes the paper and discusses some promising lines of future work. Due to space restriction the proofs are omitted. All proofs and additional examples can be found in [Echenim *et al.*, 2013].

2 Equational logic

Let $\mathcal{C}$ be a finite set of *constant symbols* (usually denoted by the letters a, b, c, ...). We assume that a total precedence $\prec$ is given on the elements of $\mathcal{C}$ (in all examples the symbols are ordered alphabetically: $a \prec b \prec c \prec \ldots$). An *atom* is an expression of the form $a \simeq b$, where $a, b \in \mathcal{C}$. Atoms are considered modulo commutativity of $\simeq$, i.e. $a \simeq b$ and $b \simeq a$ are viewed as syntactically equivalent. A *literal* is either an atom $a \simeq b$ (*positive literal*) or the negation of an atom $a \not\simeq b$ (*negative literal*). A literal l will sometimes be written $a \bowtie b$, where the symbol $\bowtie$ stands for $\simeq$ or $\not\simeq$. The literal l^c denotes the complement of l. A *clause* is a finite multiset of literals (usually written as a disjunction). As usual $\Box$ denotes the empty clause and $|C|$ is the number of literals in C. For every clause C, $\neg C$ denotes the set of clauses $\{\{l^c\} \mid l \in C\}$. For any set of clauses S, we denote by $|S|$ the cardinality of S and by $\mathrm{size}(S)$ the total size of S: $\mathrm{size}(S) \stackrel{\text{def}}{=} \Sigma_{C \in S}|C|$.

An *equational interpretation* $\mathcal{I}$ is an equivalence relation on $\mathcal{C}$. Given two constant symbols $a, b \in \mathcal{C}$, we write $a =_{\mathcal{I}} b$ if a and b belong to the same equivalence class in $\mathcal{I}$. A literal $a \simeq b$ (resp. $a \not\simeq b$) is *true* in $\mathcal{I}$ if $a =_{\mathcal{I}} b$ (resp. if $a \neq_{\mathcal{I}} b$). A clause C is *true* in $\mathcal{I}$ if it contains a literal l that is true in $\mathcal{I}$. A clause set S is *true* in $\mathcal{I}$ if all clauses in S are true in $\mathcal{I}$. We write $\mathcal{I} \models E$ and we say that $\mathcal{I}$ is a *model* of E if the expression (literal, clause or clause set) E is true in $\mathcal{I}$. For all expressions E, E', we write $E \models E'$ if every model of E is a model of E'. A *tautology* is a clause for which all equational interpretations are models and a *contradiction* is a clause that has no model. For instance, $a \not\simeq b \vee a \not\simeq c \vee b \simeq c$ is a tautology (indeed, for all equivalence relations $=_{\mathcal{I}}$, if $a =_{\mathcal{I}} b$ and $a =_{\mathcal{I}} c$, then necessarily $b =_{\mathcal{I}} c$, by transitivity), whereas $\Box$ and $a \not\simeq a$ are contradictions.

We now introduce the central notion of a prime implicate.

Definition 1 A clause C is an *implicate* of a clause set S if $S \models C$. C is a *prime implicate* of S if, moreover, C is not a tautology, and for every clause D such that $S \models D$, we have either $D \not\models C$ or $C \models D$. $\Diamond$

Example 2 Consider the clause set S:

$$\begin{array}{llll} 1 & a \simeq b \vee d \simeq a & 2 & a \simeq c \\ 3 & c \not\simeq b & 4 & c \not\simeq e \vee d \simeq e \end{array}$$

The clause $d \simeq a$ is an implicate of S, since Clauses 2 and 3 together entail $a \not\simeq b$ and thus $d \simeq a$ can be inferred from the first clause. The clause $a \not\simeq e \vee d \simeq e$ can be deduced from 4 and 2 and thus is also an implicate. But it is not prime, since $d \simeq a \models a \not\simeq e \vee d \simeq e$ (it is clear that $d \simeq a, a \simeq e \models d \simeq e$, by transitivity) but $a \not\simeq e \vee d \simeq e \not\models d \simeq a$. $\clubsuit$

The purpose of the present paper is to devise an algorithm that, given a set of clauses S, is able to compute the entire set of prime implicates of S, up to equivalence.

3 Representation of Clauses Modulo Equality

In propositional logic, detecting redundant[1] clauses is an easy task, because a clause C is a logical consequence of D iff either it is a tautology or D is a subclause of C. Thus a non-tautological clause C is redundant in a clause set iff there exists a clause $D \in S$ such that $D \subseteq C$. Furthermore, the only tautologies in propositional logic are the clauses containing two complementary literals, which is straightforward to test. The clause set S can be represented as a trie (a tree-based data-structure commonly used to represent strings [Fredkin, 1960]), so that inclusion can be tested efficiently (the literals can be totally ordered and sorted to handle commutativity). However, in equational logic, the above properties do not hold

[1] Note that the redundancy relation is defined only at the level of clauses: indeed, a clause C entailed by a clause set S is not necessarily redundant w.r.t. S in our context; for instance C can be a prime implicate of S not occurring in S.

anymore: for example the clause $a \not\simeq b \vee b \simeq c$ is a logical consequence of $a \simeq c$ but obviously $a \simeq c$ is not a subclause of $a \not\simeq b \vee b \simeq c$. Thus testing clause inclusion is no longer sufficient and representing clause sets as tries would yield many undesired redundancies: for instance the clauses $a \not\simeq b \vee b \simeq c$ and $a \not\simeq b \vee a \simeq c$ would be both stored, although they are equivalent. Our first task is thus to devise a new redundancy criterion that generalizes subsumption, together with a new way of representing clauses, that takes into account the special properties of the equality predicate. To this purpose we show how to normalize ground clauses according to the total ordering $\prec$ on constant symbols, and we introduce a new notion of *projection*.

3.1 Testing Logical Entailment

Let C be a clause. The *C-representative* of a constant a is the constant $a_{\downarrow C} \stackrel{\text{def}}{=} \min_{\prec}\{b \in \mathcal{C} \mid b \not\simeq a \models C\}$. Note that every constant has a representative, since it is clear that $a \not\simeq a \models C$. This notion extends easily to more complex expressions: $(a \bowtie b)_{\downarrow C} \stackrel{\text{def}}{=} a_{\downarrow C} \bowtie b_{\downarrow C}$ and $D_{\downarrow C} \stackrel{\text{def}}{=} \{l_{\downarrow C} \mid l \in D\}$. The expression $E_{\downarrow C}$ is called the *projection of E on C*. We write $E \equiv_C E'$ if $E_{\downarrow C} = E'_{\downarrow C}$. By definition, $\equiv_C$ is an equivalence relation and the following equivalences hold: $(a \equiv_C b) \Leftrightarrow (a \not\simeq b \models C) \Leftrightarrow (\neg C \models a \simeq b)$.

Example 3 Let $C = a \not\simeq b \vee b \not\simeq c \vee d \not\simeq e \vee a \simeq e$. We have $a \not\simeq b \models C$ and $b \not\simeq c \models C$ since both $a \not\simeq b$ and $b \not\simeq c$ occur in C. By transitivity, this implies that $a \not\simeq c \models C$, and therefore we have $a_{\downarrow C} = b_{\downarrow C} = c_{\downarrow C} = a$ (recall that constants are ordered alphabetically). Similarly, $d_{\downarrow C} = e_{\downarrow C} = d$. If f is a constant distinct from a, b, c, d, e, then $f_{\downarrow C} = f$. We have $(b \simeq e \vee a \not\simeq b)_{\downarrow C} = a \simeq d \vee a \not\simeq a$. ♣

The next proposition introduces a notion of normal form for equational clauses, which in particular permits to test efficiently whether a clause is tautological. The intuition behind this proposition is best seen by considering negations of clauses. The negation of $C = \bigvee_{i=1}^{n} a_i \not\simeq b_i \vee \bigvee_{i=1}^{m} c_i \simeq d_i$ is equivalent to the set $\neg C \stackrel{\text{def}}{=} \{a_i \simeq b_i \mid i \in [1,n]\} \cup \{c_i \not\simeq d_i \mid i \in [1,m]\}$. By definition, the relation $\equiv_C$ is the smallest equivalence relation satisfying all the equations $a_i \simeq b_i$ and $a_{\downarrow C}$ denotes the smallest representant of a modulo this relation. The relation $\equiv_C$ can be defined in a canonical way by stating that each constant a is mapped to its normal form $a_{\downarrow C}$, which is expressed by the negative literal $a \not\simeq a_{\downarrow C}$. Then each constant a can be replaced by its normal form in the positive part of the clause.

Proposition 4 *Every clause C is equivalent to the clause: $C_{\downarrow} \stackrel{\text{def}}{=} \bigvee_{a \in \mathcal{C}, a \neq a_{\downarrow C}} a \not\simeq a_{\downarrow C} \vee \bigvee_{a \simeq b \in C} a_{\downarrow C} \simeq b_{\downarrow C}$. Furthermore, C is a tautology iff $C_{\downarrow}$ contains a literal $a \simeq a$. A non-tautological clause C is in* normal form *if $C = C_{\downarrow}$ and if, moreover, all literals occur at most once in C.*

Example 5 The clause C of Example 3 is equivalent to the clause in normal form: $b \not\simeq a \vee c \not\simeq a \vee e \not\simeq d \vee a \simeq d$. Let $D \stackrel{\text{def}}{=} a \not\simeq b \vee b \not\simeq c \vee a \simeq c$; $D_{\downarrow}$ is $b \not\simeq a \vee c \not\simeq a \vee a \simeq a$, and therefore D is a tautology. ♣

We now introduce conditions that permit to design efficient methods to test if a given clause is redundant w.r.t. those stored in the database (forward subsumption) and conversely to delete from the database all clauses that are redundant w.r.t. a newly generated clause (backward subsumption).

Definition 6 Let C, D be two clauses. The clause D *eq-subsumes* C (written $D \leq_{\text{eq}} C$) iff the two following conditions hold.
- $\equiv_D \subseteq \equiv_C$ (i.e. every negative literal in $D_{\downarrow C}$ is a contradiction).
- For every positive literal $l \in D$, there exists a literal $l' \in C$ such that $l \equiv_C l'$.

If S, S' are sets of clauses, we write $S \leq_{\text{eq}} C$ if $\exists D \in S, D \leq_{\text{eq}} C$ and $S \leq_{\text{eq}} S'$ if $\forall C \in S', S \leq_{\text{eq}} C$. A clause C is *redundant* in S if either C is a tautology or there exists a clause $D \in S$ such that $D \not\equiv C$ and $D \models C$. A clause set S is *subsumption-minimal* if it contains no redundant clause. ◊

Intuitively, the test is performed by verifying that $\neg C \models \neg D$. To this purpose, we first check that all the equations in $\neg D$ are logical consequences of those in $\neg C$, which can be easily done by verifying that the relation $\equiv_D \subseteq \equiv_C$ holds. Then, we consider the negative literals in $\neg D$. Any such literal $\neg l$ can only be entailed by $\neg C$ if $\neg C$ contains a literal $\neg l'$ which can be reduced to $\neg l$ by the relation $\equiv_C$.

Example 7 Let C be the clause of Example 3. C is eq-subsumed by the clauses $a \not\simeq b \vee a \not\simeq c$, $a \not\simeq b \vee c \simeq e$ and $c \simeq d$. However, it is neither eq-subsumed by the clause $a \not\simeq d$, because $a_{\downarrow C} \neq d_{\downarrow C}$, nor by the clause $a \simeq b$, because there is no literal $l \in C$ such that $(a \simeq b)_{\downarrow C} = l_{\downarrow C}$. ♣

Theorem 8 *Let C and D be two clauses. If C is not a tautology then $D \models C$ iff $D \leq_{eq} C$.*

In the following, we will actually use a slightly more restrictive version of this criterion for redundancy elimination: we impose that the positive literals in $D_{\downarrow C}$ are mapped to *pairwise distinct* literals in $C_{\downarrow C}$. This additional restriction is necessary to prevent the factors of a clause from being redundant w.r.t. the initial clause. For example, the clause $a \simeq b \vee a \not\simeq a'$ will not be redundant w.r.t. $a \simeq b \vee a' \simeq b \vee a \not\simeq a'$, although $a \simeq b \vee a' \simeq b \vee a \not\simeq a' \models a \simeq b \vee a \not\simeq a'$.

3.2 Clausal Trees

A prime implicate generation algorithm will typically infer huge sets of clauses. It is thus essential to devise good data-structures for storing and retrieving the generated clauses, in such a way that the redundancy criterion introduced in Section 3.1 can be tested efficiently. We devise for this purpose a tree data-structure, called a *clausal tree*, specifically tailored to store sets of literals while taking into account the usual properties of the equality predicate. Similarly to tries [De Kleer, 1992], the edges of the tree are labeled by literals and the leaves are labeled either by $\square$ (representing the empty clause) or by $\emptyset$ (failure node). Each branch leading to a leaf labeled by $\square$ represents a clause defined as the disjunction of the literals labeling the nodes in the branch. Failure nodes are useful mainly to represent empty sets – in fact they can always be eliminated by straightforward simplification rules, except when the root itself is labeled by $\emptyset$.

Definition 9 A *clausal tree* is inductively defined as either $\square$, or a set of pairs (l, T') where l is a literal and T' a clausal tree.

The set of clauses represented by a clausal tree T is denoted by $\mathcal{C}(T)$ and defined inductively as follows: $\mathcal{C}(T) = \{\Box\}$ if $T = \Box$ and $\mathcal{C}(T) = \bigcup_{(l,T') \in T} \left(\bigcup_{D \in \mathcal{C}(T')} l \vee D \right)$ otherwise (note that $\mathcal{C}(\emptyset) = \emptyset$). $\Diamond$

We impose additional conditions on the clausal tree, in order to ensure that the represented clauses are in normal form and that sharing is maximal, in the sense that there are no two edges starting from the same node and labeled by the same literal. Furthermore, the literals occurring along a given branch are ordered using the usual multiset extension of $\prec$, with the additional constraints that all negative literals are strictly smaller than positive ones. More formally, we define an ordering $<$ on literals as follows.

- If l is a negative literal and l' is a positive literal, then $l < l'$.
- If l and l' have the same sign, with $l = (b \bowtie a)$, $l' = (d \bowtie c)$, $b \succeq a$ and $d \succeq c$ then $l < l'$ iff either $b \prec d$ or ($b = d$ and $a \prec c$).

Definition 10 A clausal tree T is a *normal clausal tree* if for any pair (l, T') in T, all the following conditions hold.

- There is no $T'' \neq T'$ such that $(l, T'') \in T$;
- l is not of the form $a \simeq a$ or $a \not\simeq a$, all literals occurring in T' are strictly greater than l w.r.t. $<$ and if $l = a \not\simeq b$ with $a \prec b$ then b does not occur in T'.
- T' is a normal clausal tree. $\Diamond$

It is easy to see that if T is a normal clausal tree then all the clauses in $\mathcal{C}(T)$ are in normal form.

We now introduce two algorithms for manipulating such data-structures. The first algorithm (ISENTAILED) is invoked on a clause C and a tree T, and returns *true* if and only if there exists a clause D in $\mathcal{C}(T)$ such that D eq-subsumes C. This algorithm is based on Theorem 8: it tests entailment by performing a depth-first traversal of T and attempts to project every encountered literal on C. If a literal cannot be projected, the exploration of the subtree associated to this literal is useless, so the algorithm switches to the following literal. As soon as a clause entailing C is found, the traversal halts and *true* is returned. For any expression E, $E[a := b]$ denotes the expression obtained from E by replacing all occurrences of a by b. For any clausal tree T and literal l, we denote by $l.T$ the clausal tree $l.T \stackrel{\text{def}}{=} \{(l, T)\}$. The following theorem states the properties of ISENTAILED.

Theorem 11 *The procedure* ISENTAILED *terminates in* $\mathcal{O}(\text{size}(\mathcal{C}(T)) + |C| \times |\mathcal{C}(T)|)$. *Moreover,* ISENTAILED$(C, T)$ *is true iff* $\mathcal{C}(T)$ *contains a clause* D *such that* $D \leq_{eq} C$.

The second algorithm (PRUNEENTAILED) deletes from a tree T all clauses that are eq-subsumed by C. It performs a depth-first traversal of T and attempts to project C on every clause in $\mathcal{C}(T)$, deleting those on which such a projection succeeds. As soon as a projection is identified as impossible, the exploration of the associated subtree halts and the algorithm moves on to the next clause. When every literal in C has been projected, all the clauses represented in the current subtree are entailed by C, and are therefore deleted. Afterward, the clause C can itself be added in the tree (the insertion algorithm is straightforward and thus is omitted).

Algorithm 1 ISENTAILED(C, T)

```
if T = □ then
    return true
end if
if C = □ then
    return false
end if
l_1 ← min_< {l ∈ C}
for all (l, T') ∈ T such that l ≥ l_1 do
    if l_1 = a ≄ b, with a ≻ b then
        if l = l_1 then
            if ISENTAILED(C \ {l_1}, T') then
                return true
            end if
        else if ¬(l = a ≄ c), with a ≻ c then
            if ISENTAILED(C \ {l_1}, (l.T')[a := b]) then
                return true
            end if
        end if
    else if l ∈ C then
        if ISENTAILED(C \ {l}, T') then
            return true
        end if
    end if
end for
return false
```

Theorem 12 *The procedure* PRUNEENTAILED *terminates in* $\mathcal{O}(\text{size}(\mathcal{C}(T)))$. *Moreover,* PRUNEENTAILED(C, T) *is a normal clausal tree and* $\mathcal{C}$(PRUNEENTAILED(C, T)) *contains exactly the clauses* $D \in \mathcal{C}(T)$ *such that* $C \not\leq_{eq} D$.

4 Generation of Implicates

This section addresses the problem of the generation of the implicates. We consider the inference rules below. These rules are very similar to the usual inference rules of the paramodulation calculus (see for instance [Nieuwenhuis and Rubio, 2001]). The only difference is that we allow for the replacement of arbitrary terms, provided some additional semantic conditions are attached to the conclusion. For example, the usual paramodulation rule applies on clauses of the form $C[a]$ and $a \simeq b \vee D$, yielding $C[b] \vee D$. In our context, the rule is applied on a clause $C[a']$, where a' is an arbitrary constant and the conclusion is $a \not\simeq a' \vee C[b] \vee D$. This clause can be viewed as an implication, stating that $C[b] \vee D$ holds if the condition $a \simeq a'$ is satisfied. Indeed, in this case $C[a']$ is equivalent to $C[a]$, and thus $C[b] \vee D$ can be derived by standard paramodulation.

Paramodulation (P): $$\frac{a \bowtie b \vee C \qquad a' \simeq c \vee D}{a \not\simeq a' \vee b \bowtie c \vee C \vee D}$$

Factorization (F): $$\frac{a \simeq b \vee a' \simeq b' \vee C}{a \simeq b \vee a \not\simeq a' \vee b \not\simeq b' \vee C}$$

Negative Multi-Paramodulation (M):

$$\frac{\bigvee_{i=1}^{n}(a_i \not\simeq b_i) \vee P_1 \qquad c \simeq d \vee P_2}{\bigvee_{i=1}^{n}(a_i \not\simeq c \vee d \not\simeq b_i) \vee P_1 \vee P_2}$$

Algorithm 2 PRUNEENTAILED(C, T)

if $C = \square$ **then**
  return $\emptyset$
end if
if $T = \square$ **then**
  return T
end if
$l_1 \leftarrow \min_{<} \{l_i \in C\}$
for all $(l, T') \in T$ such that $l \leq l_1$ **do**
  if $l_1 = l$ **then**
    $T'' :=$ PRUNEENTAILED$(C \setminus \{l_1\}, T')$
  else
    if $l = a \simeq b$ **then**
      $T'' :=$ PRUNEENTAILED(C, T')
    else if $l = a \not\simeq b$, with $a \succ b$
      and $\nexists c$, $l_1 = a \not\simeq c$, with $a \succ c$ **then**
      $T'' :=$ PRUNEENTAILED$(C[a := b], T')$
    end if
  end if
  $T := (T \setminus \{(l, T')\}) \cup \{(l, T'')\}$
end for
return T

We write $S \vdash C$ if C is generated from premises in S by one application of the rules P, F or M. The premises are assumed to be in normal form and the conclusion is normalized before being stored. The rule P is similar to the usual paramodulation rule, except that the unification between the terms a and a' is omitted and replaced by the addition of the literal $a \not\simeq a'$ ensuring that these terms are semantically equal. Similarly, F factorizes the literals $a \simeq b$ and $a' \simeq b'$ and adds literals ensuring that $a = a'$ and $b = b'$. The rule M corresponds to an application of a factorization rule on the negative literals $a_i \not\simeq b_i$, followed by a paramodulation step which removes these literals, while adding the conditions ensuring that $a_i = c$ and $b_i = d$.

Example 13 Consider the clauses $a \not\simeq b \vee a \not\simeq c$ and $a \simeq b$. With $n = 1$, the rule M applies on the couples of literals $(a \not\simeq b, a \simeq b)$ or $(a \not\simeq c, a \simeq c)$, the first application yielding $a \not\simeq a \vee b \not\simeq b \vee a \not\simeq c$, i.e., after normalization $a \not\simeq c$. It also applies with $n = 2$, yielding $a \not\simeq a \vee b \not\simeq b \vee a \not\simeq a \vee b \not\simeq c$, or, in normalized form, $b \not\simeq c$. ♣

A set of clauses S is *saturated* iff for every non-tautological clause C that can be derived from S using these rules, there exists a clause $C' \in S$ such that $C' \leq_{\text{eq}} C$.

The following theorem states that a saturated set S subsumes all its implicates. Therefore, if moreover S is subsumption-minimal then it contains exactly its set of prime implicates, up to equivalence.

Theorem 14 *Let S be a normalized clause set that is saturated. If $S \models C$ then $S \leq_{eq} C$.*

Putting together all the previous results, we now present the overall algorithm for prime implicates generation. It is similar to the standard "given clause" algorithm used by state-of-the-art saturation-based theorem-provers (see, e.g., [Robinson and Voronkov, 2001]). Note that the generated clauses are handled in a lazy way: rather than storing them in the clausal tree as soon as they are generated, we keep them in a clausal tree T' until they are considered for inferences. The procedure ADD(S, T) adds every clause $C \in S$ into the clausal tree T, using the previously defined procedures ISENTAILED and PRUNEENTAILED (its definition is straightforward and is omitted for the sake of conciseness). The choice of the clause in $\mathcal{C}(T')$ is heuristically guided by the cardinality of the clauses: the smallest clauses are selected with the highest priority; thus, if $\square$ is generated, then the search stops immediately.

Algorithm 3 PRIMEIMPLICATES(S)

$T := \emptyset$
% T: clausal tree used to store implicates, initially empty
$T' :=$ ADD$(S, \emptyset)$
% T': clausal tree used to store newly generated clauses
while $T' \neq \emptyset \wedge \square \notin \mathcal{C}(T)$ **do**
  Choose a clause $C \in T'$
  Remove C from T'
  if $\neg$ISENTAILED(C, T) **then**
    $T :=$ PRUNEENTAILED(C, T)
    Add C in T
    Let N be the set of clauses that can be generated from C and a premise in T
    $T' :=$ ADD(N, T')
  end if
end while
return $\mathcal{C}(T)$

Theorem 15 follows immediately from the previous results:

Theorem 15 *Let S be a set of clauses.* PRIMEIMPLICATES *terminates on S. Moreover,* PRIMEIMPLICATES(S) *is the set of prime implicates of S.*

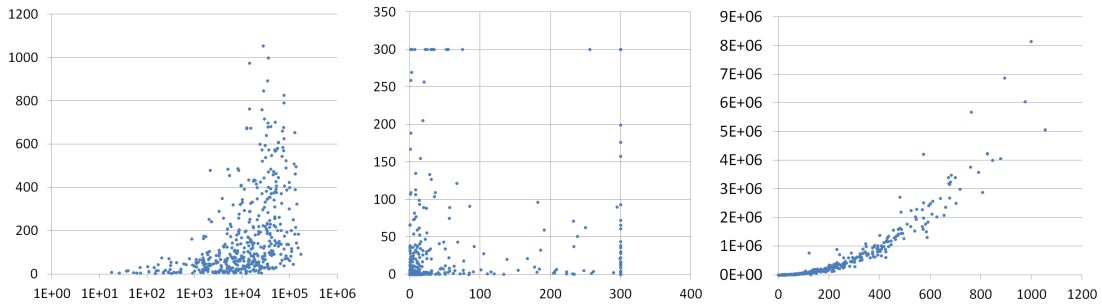
5 Experiments

We have implemented our algorithms in an Ocaml program called Kparam[2]. As far as we are aware there are two available systems for generating prime implicates of propositional formulæ. The first one is Zres [Simon and Del Val, 2001] that uses a resolution-based algorithm together with ZBDDs for storing clause sets, and the second one is ri-trie[3], which uses a decomposition method to transform the formula in a reduced implicate trie. We have chosen to compare Kparam against Zres with the "Tison" strategy, since our experiments showed that the latter performs uniformly better than the other propositional systems on the considered benchmark. Our benchmark is made of more than 500 satisfiable ground flat equational formulæ that were randomly generated. Their propositional equivalent were obtained by instantiating the transitivity[4] axiom for all constant symbols appearing in the corresponding equational formulæ. Both programs

[2] http://membres-lig.imag.fr/tourret/index.php

[3] http://www.cs.albany.edu/ritries/index.html

[4] The reflexivity and commutativity axioms are encoded directly in the transformation by orienting and simplifying the equations.



(a) Prime implicates: `Kparam`/`Zres`-tison (b) Execution time: `Kparam`/`Zres`-tison (c) `Kparam`: generated implicates/prime implicates

Figure 1: Experimental results

were run on the same machine[5] and forcibly halted after 5 minutes of execution. Our experimental results are shown in the graphs of Figure 1. Graph (1a) is a comparison (using a logarithmic scale for the X axis) of the number of prime implicates found by `Zres` for the propositional formulæ (X axis) with the one found by `Kparam` for the equivalent equational problems (Y axis). Our results indicate that the number of prime implicates is exponentially smaller in equational logic than in propositional logic. This observation is understandable if we take into account the numerous instantiations of the transitivity axiom that were necessary to translate the problems into propositional logic and the many instances of equivalent clauses that cannot be detected in a purely propositional setting. This means that the propositional output contains a lot of redundancy that has to be deleted in a post-processing step, a problem that our method averts. The results shown in Graph (1b) depict the execution time (in seconds). Note that the running time for `Zres` represented here does not include the aforementioned post-processing step. These results are somewhat less evidently in favour of `Kparam`, that is at least twice as fast 54% of the time, and globally faster 65% of the time. We have observed that the problems for which `Zres` outperforms `Kparam` are mostly those containing many unit clauses. Our system is not well-suited for this class of problems because it does not currently use equational unit propagation techniques. If we focus on problems with no initial unit clauses, then `Kparam` is faster 85% of the time (92% if simultaneous timeouts are ignored). In most cases, `Kparam` is very efficient, which is encouraging, seeing as it is only a first prototype. Graph (1c) represents the relative number of implicates (Y axis) and prime implicates (X axis) generated by `Kparam`. There is a quadratic growth of the total number of implicates generated, hence the importance of the redundancy elimination techniques from Section 3.2. This suggests that a lot of time could be gained by constraining the inference rules so as to generate less non-prime implicates.

6 Conclusion

We have devised an algorithm for generating prime implicates of clause sets defined over equations and disequations between constants, which is much more efficient than the naive approach consisting in applying the resolution calculus on the equality axioms. In particular, all the properties of the equality predicate are built-in and appropriate data-structures are used to represent clause sets. Algorithms are provided for updating such data-structures and detecting redundancy. Implicates are generated by a relaxed paramodulation rule, where equations permitting the application of the transitivity axiom are allowed to be asserted instead of being proved. The first experimental results are promising although they leave some place for improvements, at least in terms of execution time.

Future work includes the improvement of the implementation (e.g., by using a low-level programming language such as C/C++) and the refinement of the inference rules, for instance by considering ordering restrictions. The usual ordering restrictions of the superposition calculus cannot be employed in our context, because they may block the generation of some implicates, but some *partial* ordering conditions can probably be enforced while retaining completeness. Similarly, some of the literals in the clauses, more precisely the negative literals corresponding to the conditions introduced by the inference rules can be "frozen" in the sense that no further inference would be allowed within them (these literals will eventually remain – after normalization – in the considered prime implicate). Although this strategy can dismiss many inferences, its practical interest remains unclear, since the frozen literals have to be considered apart when applying the redundancy detection algorithm, which may prevent the removal of numerous clauses (this is the reason why such a strategy was not considered in our current implementation). Apart from constraining the rules, we plan to investigate other means of gaining efficiency, such as the addition of equational unit propagation techniques to handle unit clauses in a proper way, the handling of symmetric variables or the study of different strategies to select clauses. In a longer range, the extension of the presented techniques to more expressive languages (such as first-order clauses with variables and function symbols) deserves to be considered, although it raises very difficult theoretical issues: not only can termination not be enforced in general (due to well-known theoretical limitations), but also the (clausal) logical entailment relation is undecidable [Schmidt-Schauß, 1988] and even worse, not well-founded [Marquis, 1991], thus a given clause set is no longer equivalent to the conjunction of its prime implicates.

[5] Equipped with an Intel core i5-3470 CPU and 4x2 GB of RAM.

References

[De Kleer and Reiter, 1987] J. De Kleer and R. Reiter. Foundations for assumption-based truth maintenance systems: Preliminary report. In *Proc. American Assoc. for Artificial Intelligence Nat. Conf*, pages 183–188, 1987.

[De Kleer, 1992] 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.

[Echenim and Peltier, 2012] M. Echenim and N. Peltier. A Calculus for Generating Ground Explanations. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'12)*. Springer LNCS, 2012.

[Echenim *et al.*, 2013] M. Echenim, N. Peltier, and S. Tourret. An Approach to Abductive Reasoning in Equational Logic (long version). Technical report, LIG, 2013. http://membres-lig.imag.fr/peltier/EPT13.pdf.

[Fermüller *et al.*, 2001] C. G. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 25, pages 1791–1849. North-Holland, 2001.

[Fredkin, 1960] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[Jackson, 1992] Peter Jackson. Computing prime implicates incrementally. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 253–267. Springer-Verlag, 1992.

[Kean and Tsiknis, 1990] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2):185–206, 1990.

[Leitsch, 1997] A. Leitsch. *The resolution calculus*. Springer. Texts in Theoretical Computer Science, 1997.

[Marquis, 1991] Pierre Marquis. Extending abduction from propositional to first-order logic. In Philippe Jorrand and Jozef Kelemen, editors, *FAIR*, volume 535 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 1991.

[Matusiewicz *et al.*, 2009] A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 250–264, 2009.

[Mayer and Pirri, 1993] Marta Cialdea Mayer and Fiora Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of the IGPL*, 1(1):99–117, 1993.

[Mayer and Pirri, 1994] Marta Cialdea Mayer and Fiora Pirri. Propositional abduction in modal logic. *Journal of the IGPL*, 3:153–167, 1994.

[Nieuwenhuis and Rubio, 2001] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[Peirce, 1955] Charles S. Peirce. *Philosophical Writings of Peirce*. Dover Books, Justus Buchler editor, 1955.

[Ramesh *et al.*, 1997] A. Ramesh, G. Becker, and N.V. Murray. CNF and DNF considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18(3):337–356, 1997.

[Robinson and Voronkov, 2001] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.

[Rymon, 1994] R. Rymon. An se-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, 11(1):351–365, 1994.

[Schmidt-Schauß, 1988] Manfred Schmidt-Schauß. Implication of clauses is undecidable. *Theor. Comput. Sci.*, 59:287–296, 1988.

[Shanahan, 1989] Murray Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 1055–1060. Morgan Kaufmann, 1989.

[Simon and Del Val, 2001] L. Simon and A. Del Val. Efficient consequence finding. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 359–370, 2001.

[Tison, 1967] P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *Electronic Computers, IEEE Transactions on*, (4):446–456, 1967.