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 [15]) 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 for instance to infer plausible explanations of observed facts. It has many natural applications in artificial intelligence and there exists an extensive amount of research on abductive reasoning, mainly in propositional logic, with numerous applications for instance in planning [20] or truth-maintenance in knowledge bases [2]. 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

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

more difficult than merely testing whether the latter is satisfiable (note that a formula F is unsatisfiable iff `false` is an implicate of F). 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. They use either variants of the resolution rule (see, e.g., [9]), together with specific redundancy criteria and strategies ensuring efficiency [22, 8, 7, 1], or decomposition-based approaches in the spirit of the DPLL method, which compute implicates by recursively decomposing them into smaller pieces [18, 16, 11]. 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 [12, 10, 13]. In [10], 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 [4, 5]. In [12] 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 [13]. As far as we are aware, there is no published work on abductive reasoning for equational formulæ.

Abductive reasoning for equational formulæ has many applications in the fields of artificial intelligence and automated deduction. For instance, [3] proposes 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 [14] which is capable of generating, from a given set of first-order 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', and the authors rely for this purpose on a straightforward reduction to propositional logic. They use 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 (for example $a \not\simeq b \vee b \not\simeq c$, $a \not\simeq b \vee a \not\simeq c$ and $a \not\simeq c \vee b \not\simeq c$ are all equivalent). 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 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). An implementation is available.

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.

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 size(S) the total size of S: 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 [6]), 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 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,

[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.

since it is clear that $a \not\simeq a \models C$. This notion extends easily to more complex expressions: $(a \bowtie b)_{\lfloor C} \stackrel{\text{def}}{=} a_{\lfloor C} \bowtie b_{\lfloor C}$ and $D_{\lfloor C} \stackrel{\text{def}}{=} \{l_{\lfloor C} \mid l \in D\}$. The expression $E_{\lfloor C}$ is called the *projection of E on C*. We write $E \equiv_C E'$ if $E_{\lfloor C} = E'_{\lfloor C}$. 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 $a_{\lfloor C} = b_{\lfloor C} = c_{\lfloor C} = a$ (recall that constants are ordered alphabetically). Similarly, $d_{\lfloor C} = e_{\lfloor C} = d$. If f is a constant distinct from a, b, c, e, d, then $f_{\lfloor C} = f$. We have $(b \simeq e \vee a \not\simeq b)_{\lfloor C} = a \simeq d \vee a \not\simeq a$. ♣

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: the negation of a clause $C : \bigvee_{i=1}^{n} a_i \not\simeq b_i \vee \bigvee_{i=1}^{m} c_i \simeq d_i$ is equivalent to the set $\neg C = \{a_i \simeq b_i \mid i \in [1,n]\} \cup \{c_i \not\simeq d_i \mid i \in [1,m]\}$. By definition, the relation $\equiv_C$ is the smallest equivalence relation satisfying all the equations $a_i \simeq b_i$ and $a_{\lfloor C}$ denotes the smallest representant of constant 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 $a_{\lfloor C}$, which is expressed by the negative literal $a \not\simeq a_{\lfloor C}$. 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:*

$$C_{\downarrow} \stackrel{def}{=} \bigvee_{a \in \mathcal{C}, a \neq a_{\lfloor C}} a \not\simeq a_{\lfloor C} \vee \bigvee_{a \simeq b \in C} a_{\lfloor C} \simeq b_{\lfloor C}$$

Furthermore, C is a tautology iff $C_{\downarrow}$ contains a literal $a \simeq a$.

Proof. By definition of $a_{\lfloor C}$, we have $a \not\simeq a_{\lfloor C} \models C$, for every constant a. Furthermore, for every literal $a \simeq b \in C$, we have $a \simeq a_{\lfloor C}, b \simeq b_{\lfloor C}, a_{\lfloor C} \simeq b_{\lfloor C} \models a \simeq b \models C$ and therefore $C_{\downarrow} \models C$. Conversely, for every constant a, the implication $\neg C_{\downarrow} \models a \simeq a_{\lfloor C}$ holds by definition of $C_{\downarrow}$. Let l be a literal in C. If l is a negative literal $b \not\simeq a$ then we have $b \not\simeq a \models C$, hence $b_{\lfloor C} = a_{\lfloor C}$, thus $\neg C_{\downarrow} \models l^c$. If l is a positive literal $b \simeq a$ then $C_{\downarrow}$ contains a literal $a_{\lfloor C} \simeq b_{\lfloor C}$ and $a_{\lfloor C} \not\simeq b_{\lfloor C}, a \simeq a_{\lfloor C}, b \simeq b_{\lfloor C} \models l^c$, therefore we must have $\neg C_{\downarrow} \models l^c$. Consequently, $\neg C_{\downarrow} \models l^c$.

By definition, if $C_{\downarrow}$ contains $a \simeq a$ then $C_{\downarrow}$ is equivalent to `true`, and thus C is a tautology. Conversely, if $C_{\downarrow}$ contains no such literal, then we have $a \not\equiv_C b$, for every literal $a \simeq b \in C_{\downarrow}$ (since $a = a_{\lfloor C}$ and $b = b_{\lfloor C}$ by definition of $C_{\downarrow}$) thus the interpretation $\equiv_C$ falsifies every literal in $C_{\downarrow}$ (since every negative literal in $C_{\downarrow}$ is of the form $a \not\simeq a_{\lfloor C}$ and thus must be false in $\equiv_C$). Thus $C_{\downarrow}$ cannot be equivalent to `true` and C is not a tautology. ∎

Definition 5 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 6 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 $D \stackrel{\text{def}}{=} a \not\simeq b \vee b \not\simeq c \vee a \simeq c$, then $D_{\downarrow}$ 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 will 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 7 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 $D_{\restriction C}$ 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 if 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 equations in $\neg D$ are logical consequences of those in $\neg C$, which can be easily done by checking that the relation $\equiv_D \subseteq \equiv_C$ holds. Then, we consider the negative literals in $\neg D$. Such a literal $\neg l$ can only be entailed by $\neg C$ iff $\neg C$ contains a literal $\neg l'$ that can be reduced to $\neg l$ by the relation $\equiv_C$.

Example 8 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 $a_{\restriction C} \neq d_{\restriction C}$, nor by the clause $a \simeq b$, because there is no literal $l \in C$ such that $(a \simeq b)_{\restriction C} = l_{\restriction C}$. ♣

Example 9 The clause $d \simeq a$ eq-subsumes the clause $D = a \not\simeq e \vee d \simeq e$, because the clause $d \simeq a$ contains no negative literal (and thus $\equiv_{d \simeq a}$ is the identity) and the D-representatives of the literals $d \simeq e$ and $d \simeq a$ are identical.♣

Theorem 10 *Let C and D be two clauses and assume that C is not a tautology. Then $D \models C$ iff $D \leq_{eq} C$.*

Proof. Assume that $D \models C$, that C is not a tautology and that $D \not\leq_{\text{eq}} C$. If there is a negative literal $b \not\simeq a$ in D such that $b_{\restriction C} \neq a_{\restriction C}$, then $a \not\simeq b \not\models C$, hence we cannot have $D \models C$, since $b \not\simeq a \in D$. Now, consider a positive literal $b \simeq a \in D$ and assume that $b_{\restriction C} \simeq a_{\restriction C}$ does not occur in $C_{\restriction C}$. We consider the interpretation $\mathcal{I}$ such that $=_{\mathcal{I}}$ is the smallest reflexive, symmetric and transitive relation satisfying $b =_{\mathcal{I}} a$ and $d =_{\mathcal{I}} c$ for every $c, d \in \mathcal{C}$ such that $d \not\simeq c \in C$. It is clear that $\mathcal{I} \models D$, thus we must have $\mathcal{I} \models C$. Furthermore, $\mathcal{I}$ falsifies all the negative literals in C, by definition. Therefore, $\mathcal{I}$ must validate a positive literal $d \simeq c$ in C. By definition of $=_{\mathcal{I}}$ this means that there exists a sequence of constant symbols $c_1, \ldots, c_n$ such that $c_1 = d$, $c_n = c$ and for every $i \in [1, n-1]$,

we have either $c_i \not\simeq c_{i+1} \in C$ or $c_i = a$ and $c_{i+1} = b$ (or $c_{i+1} = a$ and $c_i = b$). If for every $i \in [1, n-1]$ the first condition holds, then we have $d \not\simeq c \models C$, and therefore C must be a tautology (since $d \simeq c$ occurs in C), which contradicts our hypothesis. Otherwise, we can assume, without loss of generality, that the sequence $c_1, \ldots, c_n$ is minimal (i.e., contains no repetition), so that there is *exactly* one index i satisfying the second condition. In this case we must have $d \not\simeq a \models C$ and $b \not\simeq c \models C$ (or $d \not\simeq b \models C$ and $a \not\simeq c \models C$), and thus $d_{\restriction C} = a_{\restriction C}$ and $b_{\restriction C} = c_{\restriction C}$ (or $d_{\restriction C} = b_{\restriction C}$ and $b_{\restriction C} = a_{\restriction C}$). But then, since $d \simeq c \in C$, this entails that $a_{\restriction C} \simeq b_{\restriction C}$ occurs in C which again contradicts our hypothesis.

Conversely, assume that $D \leq_{\text{eq}} C$. Let $\mathcal{I}$ be a model of D. By definition $\mathcal{I}$ validates some literal $l \in D$. If l is a negative literal $b \not\simeq a$ then we have $b_{\restriction C} = a_{\restriction C}$, and thus $b \not\simeq a \models C$. Consequently, $\mathcal{I} \models C$. If l is a positive literal $b \simeq a$ then there exists a literal $b' \simeq a' \in C$ such that ${b'}_{\restriction C} = b_{\restriction C}$ and ${a'}_{\restriction C} = a_{\restriction C}$. If $\mathcal{I} \models c \not\simeq c_{\restriction C}$ for some constant symbol $c \in \mathcal{C}$ we have $\mathcal{I} \models C$, since by definition $c \not\simeq c_{\restriction C} \models C$. Thus we can assume that $\mathcal{I} \models a \simeq a', b \simeq b'$, and since $\mathcal{I} \models b \simeq a$ we deduce that $\mathcal{I} \models b' \simeq a'$, hence that $\mathcal{I} \models C$. ∎

Remark 11 *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 $D_{\restriction C}$ are mapped to* pairwise distinct *literals in $C_{\restriction C}$. 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 $a \simeq b \vee a' \simeq b \vee a \not\simeq a' \models a \simeq b \vee a \not\simeq a'$.*

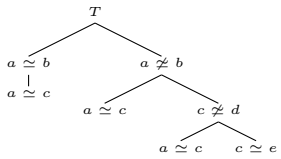
3.2 Clausal Trees

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. As in tries, the edges of the tree are labeled by literals and the leaves are labeled either by $\Box$ (representing the empty clause) or by $\emptyset$ (failure node). Each branch leading to a leaf labeled by $\Box$ 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 if the root itself is labeled by $\emptyset$.

Definition 12 A *clausal tree* is inductively defined as either $\Box$, 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 $T = \Box$
- $\mathcal{C}(T) = \bigcup_{(l,T') \in T} \left(\bigcup_{D \in \mathcal{C}(T')} l \vee D \right)$ otherwise. ◇

Note that by construction, $\mathcal{C}(\emptyset) = \emptyset$

Example 13 The structure T below is a clausal tree. There is no failure node, and for readability the labels are associated with the nodes rather than to the edges leading to them.



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

$$a \simeq b \vee a \simeq c$$
$$a \not\simeq b \vee a \simeq c$$
$$a \not\simeq b \vee c \not\simeq d \vee a \simeq c$$
$$a \not\simeq b \vee c \not\simeq d \vee c \simeq e$$

Formally, it is defined as $\{(a \simeq b, T'), (a \not\simeq b, T'')\}$, with

$$T' \stackrel{\text{def}}{=} \{(a \simeq c, \Box)\}$$
$$T'' \stackrel{\text{def}}{=} \{(a \simeq c, \Box), (c \not\simeq d, \{(a \simeq c, \Box), (c \simeq e, \Box)\})\}.$$ ♣

We impose additional conditions on clausal trees, 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 14 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. ◇

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. The tree of Example 13 satisfies these requirements, for example the constant b does not occur below the literal $a \not\simeq b$.

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. To test this entailment, the algorithm performs 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.

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
```

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 $l.T \stackrel{\text{def}}{=} \{(l, T)\}$.

Before proving that the algorithm is correct, we must verify that its requirements are always respected, namely, that for all recursive calls, the input clause is in normal form and the input tree is a normal clausal tree. For the clauses, this is obvious because no rewriting operation is ever performed, and literals are deleted from the smallest to the greatest, which is sufficient to ensure that the resulting clause remains in normal form. For the clausal trees, the proof is more involved: actually, we can remark that, although all the clauses occurring in the *initial* normal clausal tree are in normal form, this property is not preserved by recursive calls due to the fact that constants may be replaced (if an inequation $a \not\approx b$ is encountered in the clause). We thus need to introduce a slightly relaxed version of this notion.

Definition 15 A clausal tree is a *relaxed normal clausal tree* if
- for any pair $(l, T') \in T$, if l is a *negative* literal then:

- l is not of the form $a \not\simeq a$;
- for all literals l' in T', we have $l < l'$;
- if $l = a \not\simeq b$, with $a \succ b$, then for all literals l' occurring in T', the constant a does not occur in l';
- for any pair $(l, T') \in T$, T' is a relaxed normal clausal tree. $\diamond$

It is also necessary to relax the conditions on the clauses accordingly, since the clauses represented by a relaxed normal clausal tree are not necessarily in normal form:

Definition 16 A clause C is in *relaxed normal form* if $C = C_{\downarrow}$ and if, moreover, all negative literals occur at most once in C and C contains no literal of the form $a \not\simeq a$. $\diamond$

Thus, if C is in relaxed normal form, then it possibly contains several occurrences of the same positive literal, or a literal of the form $a \simeq a$.

We also need to introduce additional propositions, stating some basic properties of the clauses and clausal trees.

Proposition 17 *The two following properties hold.*
- *Let (l, T) be a relaxed normal clausal tree. If l is a* positive *literal, then all the literals occurring in T are also positive.*
- *If $T_1, \ldots, T_n$ are relaxed normal clausal trees distinct from $\Box$, then so is $\bigcup_{i=1}^{n} T_i$.*

Proof. The first item of this proposition is justified by the fact that if l and l' are literals such that $l < l'$ and l is a positive literal, then l' is also a positive literal by definition of $<$. The second item is a direct consequence of the definition of a relaxed normal clausal tree. ∎

Proposition 18 *The following properties hold:*

1. *If C and D are two non-tautological clauses containing the same negative literals then for every constant a we have $a_{\downarrow C} = a_{\downarrow D}$.*
2. *If a non-tautological clause C contains no negative literal, then for any constant a, $a_{\downarrow C} = a$.*

Proposition 19 *Let C be a clause in relaxed normal form. For any constant a, we have $a_{\downarrow C} = a$ iff for all literals l occurring in C, if l is of the form $a \not\simeq b$ then $b \succ a$.*

Proof. Let a be a constant and C be a clause in relaxed normal form; assume $a_{\downarrow C} = a$. If $l \in C$ is of the form $a \not\simeq b$ with $a \succeq b$, then by definition $b = a_{\downarrow C} = a$ and l is a contradiction; but this is impossible since C is a clause in relaxed normal form. Now assume that there exists an $l \in C$ such that l is of the form $a \not\simeq b$ with $a \succ b$, then by definition $b = a_{\downarrow C}$, thus $a \succ a_{\downarrow C}$. ∎

Proposition 20 *Consider the clauses C, D and assume that $a \not\simeq b \vee C$ is a clause in normal form, for constants a, b such that $a \succ b$. Then $D \models a \not\simeq b \vee C$ if and only if $D[a := b] \models C$.*

PROOF. Assume $D[a := b] \models C$ and let $\mathcal{I}$ be an interpretation such that $\mathcal{I} \models D$. If $a =_{\mathcal{I}} b$ then $\mathcal{I} \models a \not\simeq b \vee C$. Otherwise, we have $\mathcal{I} \models D[a := b]$ and by hypothesis $I \models C \models a \not\simeq b \vee C$.

Assume $D \models a \not\simeq b \vee C$, let $\mathcal{I} \models D[a := b]$ and consider the interpretation $\mathcal{J}$ identical to $\mathcal{I}$, except that $a =_{\mathcal{J}} b$. Then by construction $\mathcal{J} \models D[a := b]$ and $\mathcal{J} \models a \simeq b$, hence $\mathcal{J} \models D$ and $\mathcal{J} \models a \not\simeq b \vee C$. We deduce that $\mathcal{J} \models C$. Since C is in normal form, we have $C = C[a := b]$, and since constant a does not occur in C, we conclude that $\mathcal{I} \models C$. ∎

Lemma 21 proves that the requirements of the isEntailed algorithm are met at every recursive call.

Lemma 21 *Let C be a clause and T a relaxed normal clausal tree. All the trees appearing in the recursive calls of* isEntailed(C, T) *are also relaxed normal clausal trees.*

PROOF. For any $(l, T') \in T$, T' is a relaxed normal clausal tree by definition. It is also straightforward to see that $l.T'$ is a relaxed normal clausal tree. The only case to consider is when l_1 is of the form $a \not\simeq b$ with $a \succ b$, $l > l_1$ and l is not of the form $a \not\simeq c$ with $a \succ c$. We then show by induction that $(l.T')[a := b]$ (the argument of the recursive call) is a relaxed normal clausal tree.

We suppose that: $\forall (l', T'') \in T'$, $(l'.T'')[a := b]$ is a relaxed normal clausal tree. Then by Proposition 17, $T'[a := b]$ is a relaxed normal clausal tree and if l is a positive literal, then so is any literal l' occurring in T' thus $(l.T')[a := b]$ is also a relaxed normal clausal tree. If l is of the form $u \not\simeq v$ with $u \succ v$, then necessarily $u \succ a$, because $l > a \not\simeq b$ and $l \neq a \not\simeq c$ with $a \succ c$, thus $l[a := b]$ is not a contradiction. Furthermore, for all literals l' labeling an edge starting from the root of T', if l' is positive, then by definition of the order on literals, $l[a := b] < l'[a := b]$. If l' is negative, then $l' = s \not\simeq t$ with $s \succ u$ (and $s \succ t$), so $s \succ a$, hence $l[a := b] < l'[a := b]$. In addition, since u does not appear in T', it also does not appear in $T'[a := b]$ (because $u \neq b$). Since all the properties are verified, we conclude that $(l.T')[a := b]$ is a relaxed normal clausal tree. ∎

The following theorem states the main properties of isEntailed.

Theorem 22 *The procedure* isEntailed *terminates in $\mathcal{O}(\text{size}(\mathcal{C}(T)) + |C| \times |\mathcal{C}(T)|)$. If C is a clause in normal form and T is a relaxed normal clausal tree then* isEntailed(C, T) *is true iff $\mathcal{C}(T)$ contains a clause D such that $D \leq_{eq} C$.*

PROOF. We first assume that isEntailed$(C, T) = \mathit{true}$ and show by induction on size(T) that there exists a clause $D \in \mathcal{C}(T)$ such that $D \leq_{\text{eq}} C$. We examine all the cases in which isEntailed(C, T) returns *true* in their order of appearance in the algorithm.

- If $T = \Box$ then it represents the empty clause and since $\Box \leq_{\text{eq}} C$, the property holds.

- Assume $l_1 = \min_{<} \{l_i \in C\}$ is of the form $a \not\simeq b$ with $a \succ b$.

• If there exists a $(l_1, T') \in T$ such that isEntailed$(C \setminus \{l_1\}, T')$ is true, then by induction, there exists a $D \in \mathcal{C}(T')$ such that $D \leq_{\text{eq}} C \setminus \{l_1\}$. Therefore $l_1 \vee D \leq_{\text{eq}} C$ and since $l_1 \vee D \in \mathcal{C}(T)$, we have the result.

• Suppose there exists a $(l, T') \in T$ such that l is not of the form $a \not\simeq c$ with $a \succ c$ and isEntailed$(C \setminus \{l_1\}, l.T'[a := b]) = \mathit{true}$. Then by induction there exists a $D' \in \mathcal{C}(l.T'[a := b])$ such that $D' \leq_{\text{eq}} C \setminus \{l_1\}$, and therefore there exists a $D \in \mathcal{C}(l.T')$ such that $D[a := b] \leq_{\text{eq}} C \setminus \{l_1\}$. Thus we must have $D \leq_{\text{eq}} C$ and since $\mathcal{C}(l.T') \subseteq \mathcal{C}(T)$, the property is verified.

- Now assume that $l_1 = a \simeq b$ with $a \succ b$, that there exists a pair $(l, T') \in T$ with $l \geq l_1$ such that $l \in C$ and isEntailed$(C \setminus \{l\}, T')$ is true. By induction there exists a $D \in T'$ such that $D \leq_{\text{eq}} C \setminus \{l\}$. Hence $l \vee D \leq_{\text{eq}} C$, so the property is verified.

Suppose that there exists a clause $D \in \mathcal{C}(T)$ such that $D \leq_{\text{eq}} C$, we prove by induction on size(T) that isEntailed$(C,\ T) = \mathit{true}$. If $T = \Box$ then the result is clear; otherwise, D is of the form $l \vee D'$, for some $(l, T') \in T$ and $D' \in \mathcal{C}(T')$. Let $l_1 = \min_{<} \{l' \in C\}$, so that $C = l_1 \vee C'$. Note that necessarily, $l \geq l_1$. Indeed, assume this is not the case. If l_1 is of the form $a \not\simeq b$ where $a \succ b$, then l must be of the form $u \not\simeq v$, where $u \succ v$. Since $a \not\simeq b$ is minimal in C, necessarily $v_{\downarrow C} = v$ and $u_{\downarrow C} \neq v$, which means that $(u \not\simeq v)_{\downarrow C}$ cannot be a contradiction and $D \not\leq_{\text{eq}} C$. If l_1 is a positive literal then C must be a positive clause by definition of the ordering $<$, and by Proposition 18(2), $l_{\downarrow C} = l$ cannot belong to C and we cannot have $D \leq_{\text{eq}} C$.

If l_1 is of the form $a \not\simeq b$ with $a \succ b$, then there are two cases to consider. If $l = l_1$, then D is of the form $l_1 \vee D'$ and since T' is a relaxed normal clausal tree, constant a cannot occur in D' and it is straightforward to verify that $D' \leq_{\text{eq}} C'$; hence the call to isEntailed on C' and T' returns *true*. If $l > l_1$ then, since $l_{\downarrow C}$ is a contradiction, l cannot be of the form $a \not\simeq c$ with $a \succ c$ because $a_{\downarrow C} = b$ and since $a \not\simeq b$ is minimal in C, we cannot have $c_{\downarrow C} = b$. By Proposition 20, $D[a := b] \leq_{\text{eq}} C'$, and since $D[a := b] \in \mathcal{C}(l.T'[a := b])$, the call to isEntailed on C' and $l.T'[a := b]$ returns *true*.

If l_1 is a positive literal, then C must be a positive clause and by Proposition 18(2), $D_{\downarrow C} = D$. Necessarily, $l \in C$ and $D' \subseteq C \setminus \{l\}$. Therefore, the call to isEntailed on $C \setminus \{l\}$ and T' returns *true*.

We now investigate the complexity of the algorithm isEntailed. We assume that the rewriting of the constant a with the constant b performed in the recursive call isEntailed$(C \setminus \{l_1\}, l.T'[a := b])$ is not carried out by going through the whole tree $l.T'$, but simply taken into account in the following recursive calls (with a constant cost[2]). We can then estimate that in the worst case, we have one recursive call per edge in the tree T, plus one recursive call per literal

[2]because necessarily $b = b_{\downarrow C}$, thus each constant a is rewritten at most only once; and assuming a constant access to each rewritting, stored for example in a hashtable

in the clause C for each branch of T. Moreover, there are at most as many edges in T than there are literals in the clauses of $\mathcal{C}(T)$. Thus, the complexity of isEntailed(C, T) is in $\mathcal{O}(\text{size}(\mathcal{C}(T)) + |C| \times |\mathcal{C}(T)|)$. ∎

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 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

As with the previous algorithm, before proving the soundness, we must ensure that the requirements of the algorithm are met by all the recursive calls. Note that pruneEntailed(C, T) is necessarily a normal clausal tree. Indeed, it is clear that pruneEntailed does not add or modify nodes or labels in T: the only operations performed by the algorithm are replacing subtrees with empty sets and removing elements. Thus, all the conditions in Definition 14 are preserved.

The following proposition analyses the effect on the relation $\equiv_C$ of the addition of a disequation $a \not\simeq b$ into C. It is clear that this addition can only affect

the equivalence classes of the constant symbols that are already in relation with a or b:

Proposition 23 *Let C be a clause and let a, b, c, d be constant symbols. If $c \equiv_{a \not\simeq b \vee C} d$ and $c \not\equiv_C d$, then $a \not\equiv_C b$ and $\{a_{\downarrow C}, b_{\downarrow C}\} = \{c_{\downarrow C}, d_{\downarrow C}\}$. Furthermore, if $\{a_{\downarrow C}, b_{\downarrow C}\} = \{c_{\downarrow C}, d_{\downarrow C}\}$, then $c \equiv_{a \not\simeq b \vee} d$.*

PROOF. We only prove the first point, the second one being immediate. Obviously we cannot have $a \equiv_C b$, otherwise $\equiv_C$ would be identical to $\equiv_{a \not\simeq b \vee C}$. By definition of $\equiv_{a \not\simeq b \vee C}$ there exist a sequence of constant symbols $e_1, \ldots, e_n$ such that $e_1 = c$, $e_n = d$ and for all $i \in [1, n-1]$, $e_i \not\simeq e_{i+1}$ occurs in $a \not\simeq b \vee C$. W.l.o.g. we assume that this sequence is minimal, which implies that the e_i's are pairwise distinct. Then there exists at most one $i \in [1, n-1]$ such that $e_i \not\simeq e_{i+1}$ is identical to $a \not\simeq b$ (up to commutativity). Notice that such an i necessarily exists otherwise we would have $c \equiv_C d$. We have $c = e_1 \equiv_C e_i$ and $e_{i+1} \equiv_C e_n = d$. If $e_i \not\simeq e_{i+1}$ is $a \not\simeq b$, then we have $c \equiv_C a$ and $d \equiv_C b$, otherwise $e_i \not\simeq e_{i+1}$ must be $b \not\simeq a$, and we have $c \equiv_C b$ and $d \equiv_C a$. ∎

Lemma 24 *Let C be a clause in relaxed normal form and T a normal clausal tree. All the clauses arguments of a recursive call in* PRUNEENTAILED(C, T) *are in relaxed normal form.*

PROOF. Since C is in relaxed normal form, clearly $C \setminus \{l'\}$ is also in relaxed normal form for all literals l' in C. The only case that must be detailed is the recursive call PRUNEENTAILED$(C[a := b], T')$ which is invoked for $(l, T') \in T$, where l is of the form $a \not\simeq b$ with $a \succ b$; $l_1 = \min_<\{l_i \in C\}$ is not of the form $a \not\simeq c$ with $a \succ c$ and $l \leq l_1$.

If l_1 is a positive literal, then all the literals in C and $C[a := b]$ are positive. Thus by Proposition 18(2), for all $l_i \in C[a := b]$, $l_i = l_{i \downarrow C[a:=b]}$, and so $C[a := b]$ is in relaxed normal form. In the case where l_1 is a negative literal (of the form $u \not\simeq v$ with $u \succ v$), we prove by induction on the size of C that if C is in relaxed normal form, then so is $C[a := b]$. By definition, $C \setminus \{l_1\}$ is in relaxed normal form and by induction, so is $(C \setminus \{l_1\})[a := b]$. The literal $l_1[a := b]$ (denoted by l'_1 in the rest of the proof) verifies the following properties:

$l'_1 = u' \not\simeq v'$ **is not a contradiction.** Since $l_1 = u \not\simeq v$ is not a contradiction and $u \succ a$ by hypothesis, $u \neq a$, thus $u' = u$ and $v' \leq v < u$; hence $u' \neq v'$ and l'_1 cannot be a contradiction.

l'_1 **is unique in** $C[a := b]$**.** The literal l'_1 is smaller than all the positive literals in $C[a := b]$. Moreover, for any negative literal $l'_2 \in (C \setminus \{l_1\})[a := b]$, the corresponding literal $l_2 \in C \setminus \{l_1\}$ is of the form $s \not\simeq t$ (where $s \succ t$), with $s \succ u$, so $s \succ a$ and $l'_1 < l'_2$ (since $u = u'$). Thus, l'_1 is minimal in $C[a := b]$.

$l'_1 = u' \not\simeq v'$ **with** ${u'}_{\downarrow C[a:=b]} = v'$**.** Let $D = C[a := b]$ and assume ${v'}_{\downarrow D} = w$, where $w \neq v'$. Since $D[a := b] = D = C[a := b]$, by Proposition 20, $D \models a \not\simeq b \vee C$. Since $v' \neq w$, this means that $\neg C \not\models v' \simeq w$ and

$\neg C \cup \{a \simeq b\} \models v' \simeq w$. Thus by Proposition 23, either $\neg C \models v' \simeq a, w \simeq b$, or $\neg C \models v' \simeq b, w \simeq a$. But since C is in relaxed normal form, this means that C should contain either $b \not\simeq w$ or $a \not\simeq w$, and both cases are impossible since $a \not\simeq b$ is minimal in C.

In addition, since u does not appear in any literal in $C \setminus \{l_1\}$ and since $b \neq u$, for all literal $l_i \in C \setminus \{l_1\}$, $l_{i\downarrow C[a:=b]} = l_{i\downarrow (C\setminus\{l_1\})[a:=b]}$. Thus, the properties verified by induction by the literals of $(C \setminus \{l_1\})[a := b]$ are also verified in $C[a := b]$. ■

Theorem 25 *Let C be a clause in relaxed normal form and T be a normal clausal tree. Then* PRUNEENTAILED(C, T) *is a normal clausal tree that contains exactly the clauses $D \in \mathcal{C}(T)$ such that $C \not\leq_{eq} D$. Furthermore, the procedure* PRUNEENTAILED *terminates in $\mathcal{O}(\text{size}(\mathcal{C}(T)))$.*

PROOF. If D occurs in PRUNEENTAILED(C, T), then it necessarily also occurs in $\mathcal{C}(T)$, because PRUNEENTAILED(C, T) is obtained by removing some branches from T. If $C = \Box$, then we have $C \models D$ for every clause D, thus any clause in $\mathcal{C}(T)$ must be removed and in this case, the algorithm ensures that PRUNEENTAILED$(C, T) = \emptyset$. Now assume that $C \neq \Box$. We prove that $\mathcal{C}(\text{PRUNEENTAILED}(C, T)) = \{D \in \mathcal{C}(T) \mid C \not\leq_{\text{eq}} D\}$ by proving both inclusions.

Let $D \in \mathcal{C}(T)$ such that $C \leq_{\text{eq}} D$. We show by induction that $D \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T))$. If $D = \Box$ (i.e. $T = \Box$), then C must be a contradiction and since C is in relaxed normal form, $C = \Box$. Thus, PRUNEENTAILED$(C, T) = \emptyset$ and $D \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T))$. From now on, we assume that $D \neq \Box$ and $C \neq \Box$. Let $l_1 = \min_<\{l_i \in C\}$, $l = \min_<\{l' \in D\}$ and $D' = D \setminus \{l\}$. Since T is a normal clausal tree and $D \in \mathcal{C}(T)$, by definition there exists a unique normal clausal tree T' such that $(l, T') \in T$ and $D' \in \mathcal{C}(T')$. There are several cases to consider:

1. $l > l_1$, in which case no recursive call is done,
2. $l = a \not\simeq b$ and $l_1 = l$, in which case PRUNEENTAILED$(C \setminus_1\}, T')$ is invoked,
3. $l = a \not\simeq b$ and $l_1 = a \not\simeq c$, where $a \succ c$, in which case no recursive call is done,
4. $l = a \not\simeq b$, $l_1 > l$ and l_1 is not of the form $a \not\simeq c$ with $a \succ c$, in which case PRUNEENTAILED$(C[a := b], T')$ is invoked,
5. $l = a \simeq b$ and $l_1 = l$, in which case PRUNEENTAILED$(C \setminus \{l_1\}, T')$ is invoked,
6. $l = a \simeq b$ and $l_1 > l$, in which case PRUNEENTAILED(C, T') is invoked.

These cover all the possible relations between l and l_1.

1. Assume $l > l_1$. We distinguish two cases depending on the polarity of l_1:

- If $l_1 = c \not\simeq d$ with $c \succ d$, then $l_{1\restriction D}$ is a contradiction by Theorem 10, so $c_{\restriction D} = d_{\restriction D}$. But for all $l' \in D$, $l' > l_1$, and since D which is in relaxed normal form cannot contain a literal $d \not\simeq d_{\restriction D}$ which would be smaller than l_1, $d_{\restriction D} = d$. Therefore $c_{\restriction D} = d$ and by definition of a clause in normal form, $l_1 \in D$.
- If l_1 is positive, then $l_{1\restriction D} \in D_{\restriction D}$ by Theorem 10, and because D is in normal form, $l_{1\restriction D} \in D$. But since $l' > l_1$ for all $l' \in D$, D must only contain positive literals. Hence by Proposition 18(2), $l_{1\restriction D} = l_1$, and $l_1 \in D$.

Thus, in both cases, $l_1 \in D$, which is impossible since $\forall l' \in D$, $l_1 < l \leq l'$, and $C \not\leq_{\text{eq}} D$.

2. Assume $l = a \not\simeq b$ with $a \succ b$ and $l_1 = l$. In this case, PRUNEENTAILED$(C \setminus \{l_1\}, T')$ is invoked. Since $C \leq_{\text{eq}} D$, for any literal $l' \in C$ such that $l' \neq l_1$:
   - If l' is negative, then $l'_{\restriction D}$ is a contradiction. By definition of a clause in relaxed normal form, the constant a cannot appear in any literal other than l_1 in C, hence $C \setminus \{l_1\}_{\restriction D} = C \setminus \{l_1\}_{\restriction D'}$. Thus $l'_{\restriction D'}$ is also a contradiction.
   - If l' is positive then $l'_{\restriction D} \in D_{\restriction D}$. But by definition of a normal clausal tree, the positive literals of $D_{\restriction D}$ are the same as those of D, D' and $D'_{\restriction D'}$. Hence $l'_{\restriction D'} \in D'_{\restriction D'}$.

   This means that $C \setminus \{l_1\} \leq_{\text{eq}} D'$ and by induction $D' \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T'))$, thus $D \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

3. If $l = a \not\simeq b$ and $l_1 = a \not\simeq c$, where $a \succ c$, then $l_{1\restriction D} = b \not\simeq c$ is not a contradiction. Thus by Theorem 10, $C \not\leq_{\text{eq}} D$, which contradicts our hypothesis.

4. If $l = a \not\simeq b$ where $a \succ b$, $l_1 > l$ and l_1 is not of the form $a \not\simeq c$ with $a \succ c$, then PRUNEENTAILED$(C[a := b], T')$ is invoked. By Proposition 20, $C[a := b] \leq_{\text{eq}} D[a := b]$, hence $C[a := b] \leq_{\text{eq}} D'$. By induction $D' \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T'))$, hence $D \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

5. Assume $l = a \simeq b$ and $l_1 = l$. In this case, both C and D contain only positive literals, thus for any $l' \in C$ such that $l' \neq l_1$, by Proposition 18(2), $l'_{\restriction D} = l'_{\restriction D'} = l'$ and $D_{\restriction D} = D$. Furthermore, $l' \in D_{\restriction D}$, hence $l' \in D'$, and $C \setminus \{l_1\} \leq_{\text{eq}} D'$, so by induction $D' \notin \mathcal{C}(\text{PRUNEENTAILED}(C \setminus \{l_1\}, T'))$ and finally $D \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

6. If $l = a \simeq b$ and $l_1 > l$, then the same reasoning as for the previous point holds for any $l' \in C$, including l_1, thus $C \leq_{\text{eq}} D'$ and $D' \notin \mathcal{C}(\text{PRUNEENTAILED}(C, T'))$ by induction.

Now assume that $D \in \mathcal{C}(T)$ is such that $C \not\leq_{\text{eq}} D$ (this necessarily entails that $C \neq \Box$). We show by induction that $D \in \mathcal{C}(\text{PRUNEENTAILED}(C, T))$. If $D = \Box$, then $T = \Box$ and PRUNEENTAILED$(C, T) = T$, so $D \in \mathcal{C}(\text{PRUNEENTAILED}(C, T))$. As before we write $l_1 = \min_< \{l_i \in C\}$, $l =$

$\min_< \{l_i \in D\}$, $D' = D \setminus \{l\}$ and we consider the unique couple $(l, T') \in T$. By Theorem 10, there must be a literal $l' \in C$ that is not successfully projected on D. We must consider the same cases as before:

1. If $l_1 < l$, then no recursive call is done on T', and $T' \neq \emptyset$ because $D' \in \mathcal{C}(T')$, thus $D \in \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

2. Assume $l = a \not\simeq b$ with $a \succ b$ and $l_1 = l$. Clearly, $l' \neq l_1$ since $l_1 \in D$, thus $l' \in C \setminus \{l_1\}$ and l' also cannot be projected on D', because $D'_{\downarrow D'} = (D_{\downarrow D}) \setminus \{l_{\downarrow D}\}$. Hence $C \setminus \{l_1\} \not\leq_{\text{eq}} D'$ and by induction $D' \in \mathcal{C}(\text{PRUNEENTAILED}(C, T'))$. By definition, $D \in \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

3. If $l = a \not\simeq b$ and $l_1 = a \not\simeq c$, with $a \succ c$, then as seen above, $C \not\leq_{\text{eq}} D$. No recursive call is done, so $\text{PRUNEENTAILED}(C, T') = T'$, and $D \in \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

4. Assume $l = a \not\simeq b$, where $a \succ b$, $l_1 > l$ and l_1 is not of the form $a \not\simeq c$ with $a \succ c$. The constant a does not occur in l' by Proposition 18(2), thus $l'[a := b] = l'$ and $l'_{\downarrow D} = l'_{\downarrow D'}$. This permits to conclude that $l' \in C[a := b]$ and that l' cannot be projected on D' because $D'_{\downarrow D'} = (D_{\downarrow D}) \setminus \{(l_{\downarrow D})\}$; hence $C[a := b] \not\leq_{\text{eq}} D'$. By induction $D' \in \mathcal{C}(\text{PRUNEENTAILED}(C, T'))$ and so $D \in \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

5. If $l = a \simeq b$ and $l_1 = l$, then as in Point 2, $l' \neq l_1$, hence $l' \in C \setminus \{l_1\}$. Furthermore, all the literals in D are positive, thus $D'_{\downarrow D'} = D \setminus \{l\}$. This implies that l' cannot be projected on D' and by Theorem 10, $C \setminus \{l_1\}$ does not entail D'. By induction $D' \in \mathcal{C}(\text{PRUNEENTAILED}(C, T'))$ and so $D \in \mathcal{C}(\text{PRUNEENTAILED}(C, T))$.

6. If $l = a \simeq b$ and $l_1 > l$, then the proof is the same as in Point 5, except that it is possible that $l' = l_1$, thus it is C that does not entail D' instead of $C \setminus \{l_1\}$.

The two algorithms ISENTAILED and PRUNEENTAILED have a similar structure in terms of recursive calls, hence they also have a similar complexity. However, even in the worst case the recursive calls to PRUNEENTAILED always reduce the tree, which is not the case in ISENTAILED. Thus these recursive calls are not influenced by the number of literals in C, which ensure a slightly better theoretical complexity for PRUNEENTAILED than for ISENTAILED: $\mathcal{O}(\mathcal{C}(T))$ in the worst case. ∎

4 Generation of Implicates

This section addresses the problem of implicate generation. We consider the inference rules below. These rules are very similar to the usual inference rules of the paramodulation calculus (see for instance [14]). The only difference is that the replacement of arbitrary terms is allowed, provided some additional semantic conditions are attached to the conclusion. For instance, 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).

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

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

Negative Multi-Paramodulation (M):

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

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 26 The application of the rule P on $d \simeq c \vee d \simeq b$ and $d \simeq a$ yields the following clauses (among others):

$$\begin{array}{ll} d \not\simeq d \vee a \simeq c \vee d \simeq b & (\text{terms } d \text{ and } d) \\ c \not\simeq a \vee d \simeq d \vee d \simeq b & (\text{terms } c \text{ and } a) \\ b \not\simeq d \vee d \simeq c \vee d \simeq a & (\text{terms } b \text{ and } d) \end{array}$$

The clauses can be normalized afterwards: the first clause is reduced to $a \simeq c \vee d \simeq b$, the second is removed (it is a tautology) and the third one is replaced by $d \not\simeq b \vee c \simeq b \vee b \simeq a$ (since $b \prec d$). ♣

Example 27 The rule F applies on the clause $a \simeq b \vee a \simeq c \vee c \simeq d$, yielding, e.g., $a \simeq b \vee a \not\simeq a \vee b \not\simeq c \vee c \simeq d$. The normal form of the consequent is $c \not\simeq b \vee a \simeq b \vee b \simeq d$. ♣

Example 28 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, $b \not\simeq c$. ♣

Definition 29 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 $C' \leq_{\text{eq}} C$. $\diamond$

It would be tempting to remove the rule `M` (note that the case $n = 1$ in this rule is actually already covered by the rule `P`). This would make the calculus much more efficient, since the branching factor is drastically reduced; however, this also renders the calculus incomplete, as the next example proves.

Example 30 Consider the set: $S = \{c \not\simeq a \vee d \not\simeq a,\ a \simeq b \vee c \not\simeq b \vee d \not\simeq b\}$ and let $C = c \not\simeq b \vee d \not\simeq b$. It is clear that $S \models C$ (e.g. by paramodulating twice $a \simeq b$ into the first clause) and that C is a prime implicate of S. However, if `M` is forbidden, then it can be verified that S is saturated. For example, paramodulating the literal $a \simeq b$ of the second clause into the literal $c \not\simeq a$ of the first clause yields $a \not\simeq a \vee c \not\simeq b \vee d \not\simeq a \vee c \not\simeq b \vee d \not\simeq b$, and the normalized form of this clause is $c \not\simeq a \vee d \not\simeq a$, which is eq-subsumed by the first clause in S. Therefore, without rule `M`, C cannot be generated. ♣

We now prove 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. We first state the following result, that is similar in essence, but much weaker since it does not cope with redundancy elimination.

Lemma 31 *Let S be a set of clauses such that S contains every clause that can be derived from premises in S by the rules `P` or `F`. For every implicate C of S, there exists a clause $D \in S$ such that $D \models C$.*

PROOF. Let C be a non-tautological clause such that for all $D \in S$, $D \not\models C$. We begin by defining an ordering $<_C$ on literals that will permit to distinguish the literals entailing C (by ensuring that these literals are smaller than the other ones), and to order all the literals according to their projection on C. For all literals l_1, l_2, $l_1 <_C l_2$ iff one of the conditions below holds:
- $l_1 \models C$ and $l_2 \not\models C$;
- or else $l_{1\restriction C} = a_1 \bowtie b_1$, with $a_1 \succeq b_1$; $l_{2\restriction C} = a_2 \bowtie b_2$, with $a_2 \succeq b_2$ and
  - $a_1 \prec a_2$, or
  - $a_1 = a_2$ and $b_1 \prec b_2$.

Note that $<_C$ is a total ordering on atoms. This order is extended to clauses using the standard multiset extensioun. We then define an interpretation $\mathcal{I}$ that will satisfy S but not C, by induction on the ordering $<_C$. Note that $\mathcal{I}$ is constructed as a *propositional* interpretation, i.e., it maps atoms to truth values. We shall prove later that $\mathcal{I}$ actually defines an equational interpretation, i.e. an equivalence relation on $\mathcal{C}$.

Let $p_1 <_C \ldots <_C p_n$ be the set of atoms. For all $i \in \{1 \ldots n\}$, we define the propositional interpretation $\mathcal{I}_i$ as follows.

1. For any atom l such that $l_{\restriction C} = p_j$ with $j < i$, $\mathcal{I}_i \models l$ iff $\mathcal{I}_{i-1} \models l$.
2. For any atom l such that $l_{\restriction C} = p_j$ with $j > i$, $\mathcal{I}_i \not\models l$.

3. For any atom l such that $l_{\downarrow C} = p_i$, $\mathcal{I}_i \models l$ iff

   (a) either p_i is of the form $a \simeq a$,

   (b) or $p_i \notin C_\downarrow$ and there exists a clause $D \vee l' \in S$ such that:
   * ${l'}_{\downarrow C} = p_i$,
   * $D <_C l'$,
   * $\mathcal{I}_{i-1} \not\models D$.

We denote by $\mathcal{I}$ the interpretation $\mathcal{I}_n$. For all $i \in 1 \dots n$, $\mathcal{I}$ coincides with $\mathcal{I}_i$ for all literals l such that $l_{\downarrow C} = p_j$ or $l_{\downarrow C} = {p_j}^c$, where $j \leq i$. In particular, given two atoms l and l' such that $l_{\downarrow C} = {l'}_{\downarrow C}$, necessarily, either $\mathcal{I} \models l$ and $\mathcal{I} \models l'$, or $\mathcal{I} \not\models l$ and $\mathcal{I} \not\models l'$.

The proof of this lemma consists in showing that $\mathcal{I}$ is actually an equational interpretation (i.e. that it satisfies the equality axioms), and that it satisfies the formula $S \cup \neg C$. We prove first that $\mathcal{I} \not\models C$, assuming that $\mathcal{I}$ is an equational interpretation such that $\mathcal{I} \models S$, which will be proven later.

Assume that $\mathcal{I} \models C$, in this case there is a literal $l \in C$ such that $\mathcal{I} \models l$.

- If l is a positive literal, then there exists an $i \in \{1 \dots n\}$ such that $l_{\downarrow C} = p_i$. By definition of $\mathcal{I}$, either p_i is of the form $a \simeq a$, in which case $C_\downarrow$ and C are both tautologies by Proposition 4, a contradiction; or $p_i \notin C_\downarrow$, hence $l \notin C$, again a contradiction.
- Otherwise l is a negative literal and there exists an $i \in \{1 \dots n\}$ such that $l_{\downarrow C} = {p_i}^c$. In this case, p_i cannot be a tautology by Condition 3a of the definition of $\mathcal{I}$. But l is of the form $a \not\simeq b$, and since $l \models C$, necessarily $a_{\downarrow C} = b_{\downarrow C}$ and $p_i = a_{\downarrow C} \simeq b_{\downarrow C}$ is a tautology. Therefore, $\mathcal{I} \not\models C$.

We now prove that $\mathcal{I} \models S$, under the assumption that $\mathcal{I}$ is an equational interpretation. For $i = 1, \dots, n$, we consider the set $\mathcal{L}_{i,C}$ of literals l such that $p_i \not<_C l$, and define $S_{i,C} = \{D \in S \mid \forall l \in D,\, l \in \mathcal{L}_{i,C}\}$. We prove by induction that $\mathcal{I}_i$ is an equational interpretation on $\mathcal{L}_{i,C}$ such that $\mathcal{I}_i \models S_{i,C}$. Let $D \in S_{i,C}$. If $D \in S_{i-1,C}$ then by the induction hypothesis $\mathcal{I}_{i-1} \models D$, and since $\mathcal{I}_i$ coincides with $\mathcal{I}_{i-1}$ on $\mathcal{L}_{i-1,C}$, $\mathcal{I}_i \models D$. We now assume that there exists a literal $l \in D$ such that $l_{\downarrow C} \in \{p_i, {p_i}^c\}$.

1. If there are two literals l and l' in D such that $l_{\downarrow C} = p_i$ and ${l'}_{\downarrow C} = {p_i}^c$, then by construction, if $\mathcal{I}_i \not\models l$ then $\mathcal{I}_i \models l'$ and if $\mathcal{I}_i \not\models l'$ then $\mathcal{I}_i \models l$. Thus in both cases $\mathcal{I}_i \models D$.

2. If there is no literal $l \in D$ such that $l_{\downarrow C} = p_i$, then there are two possibilities to consider.

   - If there is exactly one literal $l \in D$ such that $l_{\downarrow C} = {p_i}^c$, then D is of the form $l \vee D'$, where $D' <_C l$. If $\mathcal{I}_i \models {p_i}^c$, then the result is immediate. Otherwise, either p_i is a tautology, in which case $l_{\downarrow C}$ is a contradiction and by Theorem 10, $l \models C$, which, by definition of $<_C$, entails that $l' \models C$ for every $l' \in D'$, so that $D \models C$, a contradiction; or p_i is not a tautology, which entails that $p_i \notin C_\downarrow$. In this case, there exists a clause $D'' \vee l' \in S$ such that ${l'}_{\downarrow C} = p_i$, $D'' <_C l'$ and $\mathcal{I}_{i-1} \not\models D''$. Assume $l = a \not\simeq b$ and

$l' = a' \simeq b'$ with $a_{\downarrow C} = a'_{\downarrow C}$ and $b_{\downarrow C} = b'_{\downarrow C}$, and let $E = a \not\simeq a' \vee b \not\simeq b' \vee D' \vee D''$. Note that $E <_C p_i$ because $a \not\simeq a', b \not\simeq b' \models C$ and $D', D'' <_C p_i$. By inference rule P, $l \vee D', l' \vee D'' \vdash E$, thus $E \in S_{i-1,C}$. By the induction hypothesis $\mathcal{I}_i \models E$ but $\mathcal{I}_i \not\models a \not\simeq a', b \not\simeq b', D', D''$ by construction, so that $\mathcal{I}_i \not\models E$, a contradiction.

- If there are several literals $l \in D$ such that $l_{\downarrow C} = {p_i}^c$, then the same disjunction of cases can be applied as with only one such literal, with the difference that in the second case, the rule P is applied several times until a clause E of the form $l_1 \vee \ldots \vee l_k \vee D' \vee \ldots \vee D' \vee D'' \vee \ldots \vee D''$ where $l_1, \ldots, l_k$ are negative literals entailing C is generated. The same contradiction can then be raised on E.

3. If there is no literal $l \in D$ such that $l_{\downarrow C} = {p_i}^c$, then there are again two possibilities to consider.
   - If there is exactly one literal $l \in D$ such that $l_{\downarrow C} = p_i$, then D is of the form $D' \vee l$, where $D' <_C p_i$. If $\mathcal{I}_{i-1} \models D'$, then by definition $\mathcal{I}_i \models D$. Otherwise, if $p_i \in C_\downarrow$, then $l \models C$ by Theorem 10 and by definition of $<_C$, for all literals $l' \in D'$, necessarily $l' \models C$, so that $D \models C$, which contradicts our hypothesis. Thus $p_i \notin C_\downarrow$ and l verifies Condition 3b of the definition of $\mathcal{I}$, hence $\mathcal{I}_i \models l$ and $\mathcal{I}_i \models D$.
   - Assume there are two maximal literals $l, l' \in D$ such that $l_{\downarrow C} = {l'}_{\downarrow C} = p_i$. Then D is of the form $l \vee l' \vee D'$, where $D' <_C l$ and $D' <_C l'$. The proof is similar if there are more than two maximal atoms, by applying several time the rule F instead of just once. Suppose l is of the form $a \simeq b$ and l' is of the form $a' \simeq b'$, where $a_{\downarrow C} = {a'}_{\downarrow C}$ and $b_{\downarrow C} = {b'}_{\downarrow C}$. Let $E = a \not\simeq a' \vee b \not\simeq b' \vee l \vee D'$. By inference rule F, $D \vdash E$. Thus, $E \in S$ and the previous case proves that $\mathcal{I}_i \models l$ and $\mathcal{I}_i \models D$.

This proves that $\mathcal{I}_i \models S_{i,C}$, there remains to prove that the restriction of $\mathcal{I}_i$ to $\mathcal{L}_{i,C}$ is an equational interpretation. Let $l = a \simeq b$ be a literal such that $l_{\downarrow C} = p_i$.

Reflexivity: if l is a tautology, then so is p_i and by construction $\mathcal{I}_i \models l$.

Commutativity: since $a \simeq b$ and $b \simeq a$ are assumed to be identical, commutativity is naturally respected by $\mathcal{I}_i$.

Transitivity: assume $\mathcal{I}_i \models a \simeq b$ and $\mathcal{I}_i \models a \simeq c$, where $a \simeq c \in \mathcal{L}_{i,C}$, we prove that $\mathcal{I}_i \models b \simeq c$, provided that $b \simeq c \in \mathcal{L}_{i,C}$. There are several cases to consider, depending on which of $(a \simeq b)_{\downarrow C}$ or $(a \simeq c)_{\downarrow C}$ is a tautology.

1. If $b_{\downarrow C} = a_{\downarrow C} = c_{\downarrow C}$, then $(b \simeq c)_{\downarrow C} = p_i$ is a tautology and by construction, $\mathcal{I}_i \models b \simeq c$.
2. Assume $b_{\downarrow C} = a_{\downarrow C}$ and $(a \simeq c)_{\downarrow C}$ is not a tautology. Then there is a $j \leq i$ such that $(a \simeq c)_{\downarrow C} = p_j$, and there exists a clause $D \vee a' \simeq c' \in S$ such that $(a \simeq c)_{\downarrow C} = (a' \simeq c')_{\downarrow C}$, $D <_C (a' \simeq c')$ and $\mathcal{I}_{j-1} \not\models D$. By construction $\mathcal{I}_j \models a \simeq c$, and since $\mathcal{I}_i$ and $\mathcal{I}_j$ coincide on $\mathcal{L}_{j,C}$,

we deduce that $\mathcal{I}_i \models a \simeq c$. But $(b \simeq c)_{\downarrow C} = (a \simeq c)_{\downarrow C}$, therefore $\mathcal{I}_i \models b \simeq c$.

3. The same reasoning proves the result if $c_{\downarrow C} = a_{\downarrow C}$ and $(a \simeq b)_{\downarrow C}$ is not a tautology.
4. Assume neither $(a \simeq b)_{\downarrow C}$ nor $(a \simeq c)_{\downarrow C}$ is a tautology. Then there exists a clause $D \vee d \simeq e \in S_{i,C}$ such that $(a \simeq b)_{\downarrow C} = (d \simeq e)_{\downarrow C}$, $D <_C (d \simeq e)$ and $\mathcal{I}_{i-1} \not\models D$. W.l.o.g., we assume that $a_{\downarrow C} = d_{\downarrow C}$ and $b_{\downarrow C} = e_{\downarrow C}$. Similarly, for some $j \leq i$, there exists a clause $D' \vee d' \simeq f$ in $S_{j,C}$ such that $(a \simeq c)_{\downarrow C} = (d' \simeq f)_{\downarrow C}$, $D' <_C (d' \simeq f)$ and $\mathcal{I}_{j-1} \not\models D'$. W.l.o.g., we assume that $a_{\downarrow C} = {d'}_{\downarrow C}$ and $c_{\downarrow C} = f_{\downarrow C}$. Let $E = d \not\simeq d' \vee e \simeq f \vee D \vee D'$. If $E \notin S_{i,C}$, then since $d \not\simeq d' \models C$ and $D \vee D' \in S_{i-1,C}$, necessarily $p_i <_C (e \simeq f)_{\downarrow C}$ and there is nothing to prove. Otherwise, Since $\mathcal{I}_i \models S_{i,C}$, we deduce that $\mathcal{I}i \models E$; and since $\mathcal{I}i \not\models D, D', d \not\simeq d'$, necessarily $\mathcal{I}i \models e \simeq f$, and $\mathcal{I}i \models b \simeq c$. ∎

Theorem 34 states a more powerful result than Lemma 31, namely that any implicate of a saturated set S is eq-subsumed by a clause in S. The proof of Theorem 34, requires the handling of redundancy elimination. We first show that a disjunction of literals that are all eq-subsumed by a clause D can be factorized into a clause that is itself eq-subsumed by D.

Proposition 32 *Let C, D be two clauses. Assume that for every literal $l \in C$, we have $l \leq_{eq} D$. There exists a clause C' derivable by factorization from C such that $C' \leq_{eq} D$.*

PROOF. By definition, for all negative literals $a \not\simeq b \in D$, we have $a \equiv_D b$, . Moreover, there exists a function γ mapping the positive literals in C to positive literals in D such that $\forall l \in C, l \equiv_D \gamma(l)$. Note that if γ is not injective, then we do not have $C \leq_{\text{eq}} D$. Suppose that C is of the form $a \simeq b \vee c \simeq d \vee C''$, where $(a \simeq b) \equiv_D (c \simeq d)$. W.l.o.g., we may assume that $a \equiv_D c$ and $b \equiv_D d$. Then inference rule `F` applied to C generates $a \not\simeq c \vee b \not\simeq d \vee a \simeq b \vee C''$. This clause satisfies the same requirements as C and contains one less positive literal. By induction, we deduce that there exists a clause C' derivable by factorization from C such that $C' \leq_{\text{eq}} D$. ∎

The next lemma deals with inferences applied to positive literals only. It shows that any sequence of such inferences on a set of clauses S' can be "simulated" by applying inference rules on any set $S \leq_{\text{eq}} S'$.

Lemma 33 *Let S be a set of clauses that is saturated. If $S \leq_{eq} S'$ and if C is a clause deduced from clauses in S' by a sequence of applications of the Factorisation or Paramodulation rules into positive literals, then $S \leq_{eq} C$.*

PROOF. The proof is done by induction on the number of inference steps. If C occurs in S' then the proof is immediate since $S \leq_{\text{eq}} S'$. Otherwise, let D be the first clause generated in the derivation leading to C. D must be deduced from clauses in S' by applying either the rule `P` into a positive literal or the rule `F`.

We consider only the case of the rule P, the case of the rule F is similar. Assume that $D = a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$ is generated by P, from clauses $a \simeq b \vee P_1$ and $c \simeq d \vee P_2$. We prove that $S \leq_{\text{eq}} a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$. By definition, S contains a clause eq-subsuming $a \simeq b \vee P_1$. If this clause eq-subsumes P_1, then it also eq-subsumes $a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$. The case where S contains a clause eq-subsuming P_2 is symmetrical. Otherwise, S contains a clause of the form $a' \simeq b' \vee P'_1$, where $P'_1 \leq_{\text{eq}} P_1$ (since $\equiv_{P'_1}$ and $\equiv_{a' \simeq b' \vee P'_1}$ are identical and since no positive literal in P'_1 can be mapped to $a \simeq b$) and $(a \simeq b) \equiv_{P_1} (a' \simeq b')$; and a clause $c' \simeq d' \vee P'_2$, with $(c' \simeq d') \equiv_{P_2} (c \simeq d)$ and $P'_2 \leq_{\text{eq}} P_2$. The Paramodulation rule applied to both clauses yields $a' \simeq d' \vee b' \not\simeq c' \vee P'_1 \vee P'_2$, and this clause eq-subsumes $a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$. Since S is saturated, we have $S \leq_{\text{eq}} a' \simeq d' \vee b' \not\simeq c' \vee P'_1 \vee P'_2$, hence $S \leq_{\text{eq}} a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2$. Thus, $S \leq_{\text{eq}} S' \cup \{a \simeq d \vee b \not\simeq c \vee P_1 \vee P_2\}$ and by the induction hypothesis, $S \leq_{\text{eq}} C$. ∎

Theorem 34 *Let S be a normalized clause set that is saturated. If $S \models C$ then $S \leq_{eq} C$.*

Proof. Assume that $S \models C$ and let $S' = \{D' \mid \exists D \in S, D \leq_{\text{eq}} D'\}$. Note that $S \subseteq S'$ and $S \leq_{\text{eq}} S'$; thus $S \equiv S'$ and $S \models C$ if and only if $S' \models C$. We prove that all clauses that can be derived from S' by P or F are eq-subsumed by S. If this is the case then S also eq-subsumes the closure S'' by inference rules P and F of S', and since $S'' \leq_{\text{eq}} C$ by Lemma 31, we will have the result. Since $S \leq_{\text{eq}} S'$, by Lemma 33, we already know that S eq-subsumes all clauses that can be obtained from clauses in S' by applying the Factorization or Paramodulation rule into positive literals. Thus we only consider the case of the application of the Paramodulation rule into negative literals.

Let $a \not\simeq b \vee P_1$, $c \simeq d \vee P_2$ be two clauses in S', for which the Paramodulation rule generates $Q = a \not\simeq c \vee b \not\simeq d \vee P_1 \vee P_2$. We prove that $S \leq_{\text{eq}} Q$. The same reasoning as in the proof of Lemma 33, can be used to show that either $S \leq_{\text{eq}} P_2$, in which case $S \leq_{\text{eq}} Q$, or S contains a clause $c' \simeq d' \vee P'_2$ such that $c \equiv_{P_2} c'$, $d \equiv_{P_2} d'$ and $P'_2 \leq_{\text{eq}} P_2$. We distinguish two cases involving $a \not\simeq b \vee P_1$.

- Assume that S contains a clause P'_1 such that $P'_1 \leq_{\text{eq}} a \not\simeq b \vee Q$ and $\equiv_{P'_1} \subseteq \equiv_Q$. By definition, there exists an injective function γ mapping the positive literals in P'_1 to the positive literals in $a \not\simeq b \vee Q$ such that for every positive literal $l \in P'_1$, $l \equiv_{a \not\simeq b \vee Q} \gamma(l)$. Let D be the disjunction of positive literals $l \in P'_1$ such that $l \not\equiv_Q \gamma(l)$, and let D' be the disjunction of the literals $\gamma(l)$ for $l \in D$. Note that $D' \subseteq Q$, since $a \not\simeq b$ is negative. By definition, every negative literal in P'_1 eq-subsumes Q, since $\equiv_{P'_1} \subseteq \equiv_Q$ by hypothesis, and for all positive literals l in $P'_1 \setminus D$, we have $l \equiv_Q \gamma(l)$, hence $l \leq_{\text{eq}} Q$. Thus P'_1 is of the form $Q'_1 \vee D$, where $Q'_1 \leq_{\text{eq}} Q$.

  By construction $D \equiv_{a \not\simeq b \vee Q} D'$, thus for every constant symbol g occurring in D, there exists a constant g' occurring at the same position in D' such that $g \equiv_{a \not\simeq b \vee Q} g'$. We consider the clause D'' obtained from D by replacing every constant symbol g by a constant g'' chosen as follows:

- If $g \not\equiv_Q g'$ and $g \equiv_Q a$ then $g'' = d'$.

- If $g \not\equiv_Q g'$ and $g \equiv_Q b$ then $g'' = c'$.
- Otherwise $g'' = g$.

We show that $g'' \equiv_Q g'$ for every constant g in D. Assume that this property is falsified for some constant g. If $g \equiv_Q g'$ then $g'' = g$ by definition of g'', a contradiction. If $g \equiv_Q a$ and $g \not\equiv_Q g'$, then by Proposition 23 $g' \equiv_Q b$, since $g \equiv_{a \not\simeq b \vee Q} g'$ by hypothesis. But then $g'' = d'$, and since $d' \equiv_{P_2} d$, we deduce that $d' \equiv_Q d \equiv_Q b \equiv_Q g'$, which contradicts our initial assumption. The proof is similar if $g \equiv_Q b$ and $g \not\equiv_Q g'$. Otherwise, we must have $g \not\equiv_Q a$, $g \not\equiv_Q b$ and $g = g''$, which is impossible by Proposition 23, since otherwise we would have $g \not\equiv_{a \not\simeq b \vee Q} g'$. Therefore $g'' \equiv_Q g'$ for every constant g in D, and $D'' \equiv_Q D'$. Since $D \subseteq Q$, this entails that $D'' \leq_{\text{eq}} Q$.

The Paramodulation of $c' \simeq d' \vee P'_2$ into $Q'_1 \vee D$ generates a clause E of the form $Q'_1 \vee D'' \vee F \vee P'_2 \vee \cdots \vee P'_2$, where F is a disjunction of disequations of one of the following forms:

- $g \not\simeq c'$ with $g \equiv_Q a$ and $g \not\equiv_Q g'$
- or $g \not\simeq d'$ with $g \equiv_Q b$ and $g \not\equiv_Q g'$

Since $c' \equiv_Q c \equiv_Q a$ and $d' \equiv_Q d \equiv_Q b$, it is clear that $F \leq_{\text{eq}} Q$. Since $P'_2 \leq_{\text{eq}} Q$, $Q'_1 \leq_{\text{eq}} Q$ and $D'' \leq_{\text{eq}} Q$, by Proposition 32, the clause E can be factorized into a clause $F' \leq_{\text{eq}} Q$. By Lemma 33 $S \leq_{\text{eq}} F'$, hence $S \leq_{\text{eq}} Q$.

- Assume that S contains no clause P'_1 such that $P'_1 \leq_{\text{eq}} a \not\simeq b \vee Q$ and $\equiv_{P'_1} \subseteq \equiv_Q$. Since $S \leq_{\text{eq}} S'$, S must contain a clause $P'_1 \leq_{\text{eq}} a \not\simeq b \vee P_1$, and we may assume that $\equiv_{P'_1} \not\subseteq \equiv_{P_1}$ (otherwise we would have $\equiv_{P'_1} \subseteq \equiv_Q$ and $P'_1 \leq_{\text{eq}} a \not\simeq b \vee Q$, hence we would be back in the previous case). P'_1 is of the form $\bigvee_{i=1}^{n}(e_i \not\simeq f_i) \vee P''_1$, where $\equiv_{P''_1} \subseteq \equiv_{P_1}$, and for every $i \in [1, n]$, we have $e_i \equiv_{a \not\simeq b \vee P_1} f_i$ but $e_i \not\equiv_{P_1} f_i$. By Proposition 23, we know that for every $i \in [1, n]$, we have either $e_i \equiv_{P_1} a$ and $f_i \equiv_{P_1} b$ or $e_i \equiv_{P_1} b$ and $f_i \equiv_{P_1} a$. By commutativity, we may assume that we always have $e_i \equiv_{P_1} a$ and $f_i \equiv_{P_1} b$.

The rule `M` applied to P'_1 and $c' \simeq d' \vee P'_2$ generates $R = \bigvee_{i=1}^{n}(e_i \not\simeq c' \vee d' \not\simeq f_i) \vee P''_1 \vee P'_2$. We have $c' \equiv_{P_2} c$ and $c \equiv_{a \not\simeq c} a$, hence by Proposition 23, for all $i \in [1, n]$, $c' \equiv_{a \not\simeq c \vee P_1 \vee P_2} e_i$ and $c' \equiv_Q e_i$. Similarly, for every $i \in [1, n]$, $d' \equiv_{b \not\simeq d \vee P_1 \vee P_2} f_i$ hence $d' \equiv_Q f_i$. Since $P''_1 \leq_{\text{eq}} a \not\simeq b \vee P_1$ by definition and $P'_2 \leq_{\text{eq}} P_2$ we deduce that $R \leq_{\text{eq}} a \not\simeq b \vee Q$. Furthermore, since $\equiv_{P''_1} \subseteq \equiv_{P_1}$, we deduce that $\equiv_R \subseteq \equiv_Q$. Since S is saturated, S contains a clause $R' \leq_{\text{eq}} R$. We have $\equiv_{R'} \subseteq \equiv_Q$ and $R' \leq_{\text{eq}} a \not\simeq b \vee Q$, and the previous case proves the result.

■

Putting together the previous results, we 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., [17]). 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, they are kept in a clausal tree T' until they are considered for inferences. The procedure $\textsc{Add}(S, T)$ adds every clause $C \in S$ into the clausal tree T, using the previously defined procedures isEntailed and pruneEntailed (its definition is straightforward). 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)

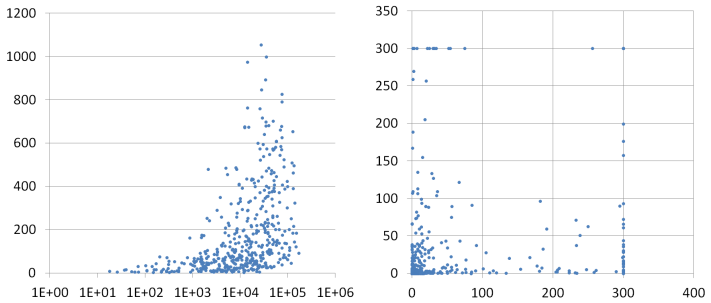
$T := \emptyset$
% *T is the clausal tree used to store the implicates, it is initially empty*
$T' :=$ ADD$(S, \emptyset)$
% *T' is the clausal tree used to store the newly generated clauses*
while $T' \neq \emptyset \wedge \square \notin \mathcal{C}(T)$ **do**
  Choose a clause $C \in T'$
  Remove C 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 35 *Let S be a set of clauses.* PRIMEIMPLICATES *terminates on S. Moreover,* PRIMEIMPLICATES(S) *is the set of prime implicates of S.*

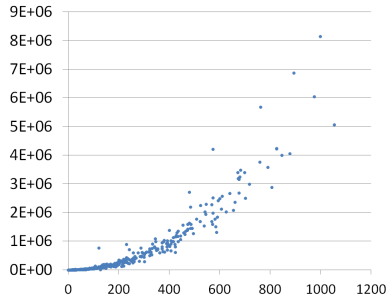
5 Experiments

We have implemented our algorithms in an Ocaml program called `K-param`[3]As far as we are aware there are two available systems for generating prime implicates of propositional formulæ. The first one is `Zres` [21] that uses a resolution-based algorithm together with ZBDDs for storing clause sets, and the second one is `ri-trie`[4], which uses a decomposition method to transform the formula in a reduced implicate trie. We have chosen to compare K-param 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[5] axiom for all constant symbols appearing in the corresponding equational formulæ. Both programs were run on the same machine[6] and forcibly halted after 5 minutes of execution. Our experimental results are shown in the graphs of Figure 1. Graph

[3] `http://membres-lig.imag.fr/tourret/index.php?&slt=tools`
[4] `http://www.cs.albany.edu/ritries/index.html`
[5] The reflexivity and commutativity axioms are encoded directly in the transformation by orienting and simplifying the equations.
[6] equipped with an Intel core i5-3470 CPU and 4x2 GB of RAM



(a) Prime implicates: K-param/Zres-tison (b) Execution time: K-param/Zres-tison



(c) K-param: generated implicates/prime implicates

Figure 1: Experimental results

(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 K-param 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) concern 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 K-param, 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 K-param 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 K-param is faster 85% of the time

(92% if simultaneous timeouts are ignored). In most cases, K-param 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 K-param. 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 [19] and even worse, not well-founded [10], thus a given clause set is no longer equivalent to the conjunction of its prime implicates.

References

[1] 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.

[2] 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.

[3] 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.

[4] C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution Methods for the Decision Problem*. LNAI 679. Springer, 1993.

[5] 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.

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

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

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

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

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

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

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

[13] M. C. Mayer and F. Pirri. Propositional abduction in modal logic. *Journal of the IGPL*, 3:153–167, 1994.

[14] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

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

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

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

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

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

[20] M. 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.

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

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