Journal of Artificial Intelligence Research 60 (2017) 827-880 Submitted 02/17; published 12/17

Prime Implicate Generation in Equational Logic

Mnacho Echenim MNACHO.ECHENIM@UNIV-GRENOBLE-ALPES.FR
Nicolas Peltier NICOLAS.PELTIER@UNIV-GRENOBLE-ALPES.FR
Univ. Grenoble Alpes, CNRS,
Grenoble INP, LIG,
F-38000 Grenoble France

Sophie Tourret SOPHIE.TOURRET@MPI-INF.MPG.DE
Max-Planck-Institut für Informatik,
Saarland Informatics Campus,
Campus E1 4, 66123 Saarbrücken, Germany

Abstract

We present an algorithm for the generation of prime implicates in equational logic, that is, of the most general consequences of formulæ containing equations and disequations between first-order terms. This algorithm is defined by a calculus that is proved to be correct and complete. We then focus on the case where the considered clause set is ground, i.e., contains no variables, and devise a specialized tree data structure that is designed to efficiently detect and delete redundant implicates. The corresponding algorithms are presented along with their termination and correctness proofs. Finally, an experimental evaluation of this prime implicate generation method is conducted in the ground case, including a comparison with state-of-the-art propositional and first-order prime implicate generation tools.

1. Introduction

We tackle the problem of generating the prime implicates of an equational formula. Formally, an implicate of a formula (or set of clauses) ϕ is a clause C that is a logical consequence of ϕ, written $\phi \models C$. This implicate is prime if for all implicates D such that $D \models C$, we have $C \models D$. In other words, prime implicates are the most general clausal consequences of a formula and in particular, $\square$ is an implicate of ϕ if and only if ϕ is unsatisfiable.

Prime implicate generation is a more difficult problem than satisfiability checking, and has many natural applications in artificial intelligence and system verification, such as diagnosis (De Kleer, 1992), debugging (Echenim & Peltier, 2012) or knowledge representation (Darwiche & Marquis, 2002). In propositional logic, the computation of prime implicates plays a central role in the minimization of boolean functions (Quine, 1955). Such a relation does not exist in more expressive logics, but prime implicate computation also has numerous applications related to abductive reasoning. Indeed, the negation of an implicate I of a formula ϕ can be viewed as a sufficient condition ensuring that ϕ is false. If $\phi = KB \wedge \neg Conc$, where KB is a knowledge base and $Conc$ is a conclusion to be proven, then $\neg I$ can be seen as a sufficient condition ensuring that $Conc$ holds, or as a set of "missing hypotheses" that can be used to derive $Conc$ from the axioms in KB. In particular, abductive reasoning can be very useful for debugging verification problems. Indeed, mistakes frequently arise when writing logical specifications, ranging from obvious (but annoying) typing errors, to forgotten cases or more fundamental issues. The effect of these errors is usually that the prover fails to find a proof, leaving to the user the burden of understanding where the problem comes from. In some cases, counter-examples can be automatically computed, and these counter-examples can then be analyzed to locate the source of the error. However, when dealing with very expressive logics, such counter-examples may not be very informative, because they are too complex and not focused enough on the relevant part of the problem. In some cases, they can even be impossible to find, for instance if the considered knowledge base has no finite models. On the other hand, pointing out missing hypotheses may allow the user to quickly grasp where the problem comes from, and may even suggest hints to correct the specification.

1.1 Roadmap

The rest of the paper is structured in the following way. In Section 2, usual definitions in first-order equational logic are briefly reviewed. In Section 3, a representation of ground[1] equational clauses up to equivalence modulo equality is defined and techniques are devised for testing logical entailment efficiently (by duality, the results also apply to sets of unit ground equational clauses). In Section 4, a calculus is presented to generate implicates of equational clauses. For the sake of generality, this calculus is defined for non-ground clauses, although the following sections focus on ground clauses. The defined calculus is an extension of the Superposition calculus (Bachmair & Ganzinger, 1994), which is the most advanced and successful proof procedure for first-order logic with equality. The underlying idea behind this calculus is based on so-called *Assertion rules* that add new hypotheses into the considered set on demand, in a controlled way. These hypotheses are collected and attached to the clauses as constraints; if the empty clause is generated, then its constraint corresponds to the negation of an implicate of the original formula. The use of constraints also permits to control the class of implicates that are generated: for example, it is possible to use this calculus to generate implicates up to a fixed size, or to generate only the positive implicates. This is a crucial feature in practice since the number of prime implicates is usually huge. In Section 5, data structures are defined to store sets of *ground* constrained clauses efficiently. We devise algorithms based on the results of Section 3 for detecting redundant constrained clauses, i.e., for checking whether a newly generated clause is redundant and to remove all the previously generated clauses it entails, and we prove that these algorithms are sound and complete. In Section 6, experimental comparisons with all available systems for generating implicates in propositional and first-order logic are provided, showing evidence of the relevance of our approach. Finally, in Section 7, potential applications and relevant related work are discussed, and lines of future work are suggested.

The present paper is a thoroughly expanded version of a conference paper (Echenim, Peltier, & Tourret, 2015). The procedure and the completeness proof have been revised and extended in order to handle arbitrary first-order clauses, whereas only ground clauses were considered in the conference paper. Furthermore, the procedure is now parameterized by a set of constraints, denoting abducible hypotheses, and deductive completeness is directly established w.r.t. this set of constraints. Finally, detailed proofs of the correction of the redundancy detection algorithms are now provided. Note that, while the calculus and the

1. I.e., without variables.

theoretical results handle arbitrary first-order clauses as input, the data-structures and algorithms for efficient redundancy detection are defined only for ground clauses and the experimental evaluation is performed only for such clauses.

2. Clausal First-Order Logic with Equality

This section contains definitions and notations related to first-order logic with equality.

2.1 Equational Clausal Logic

We introduce the standard syntax and semantics of equational clausal logic used in the paper.

2.1.1 Syntax

In this section, we briefly review usual definitions on equational clausal logic. The definitions are identical or very similar to those used in standard textbooks (see, e.g., Baader & Nipkow, 1998; Nieuwenhuis & Rubio, 2001).

We consider first-order logic with equality, in which formulæ admit equality (denoted by $\simeq$) as a unique predicate[2]. For $n \geq 0$, Σ_n denotes a *signature* of function symbols of arity n, and we let $\Sigma \stackrel{\text{def}}{=} \bigcup_{n=0}^{\infty} \Sigma_n$. The elements of Σ_0 are constant symbols. Function symbols of arity strictly greater than 0 are usually denoted by f or g and constant symbols by a, b, c or d. Let $\mathcal{V}$ be a set of *variables*, usually denoted by x, y or z, disjoint from Σ. The notation $\mathfrak{T}_{\mathcal{V}}(\Sigma)$ stands for the set of *terms* over Σ, defined inductively as follows:

- $\mathcal{V} \cup \Sigma_0 \subseteq \mathfrak{T}_{\mathcal{V}}(\Sigma)$;
- if $t_1, \ldots, t_n \in \mathfrak{T}_{\mathcal{V}}(\Sigma)$, $f \in \Sigma_n$ and $n > 0$ then $f(t_1, \ldots, t_n) \in \mathfrak{T}_{\mathcal{V}}(\Sigma)$.

Terms are usually denoted by s, t, u, v, w. A *position* is a (possibly empty) finite sequence of natural numbers. The empty position is denoted by ε and the concatenation of two positions p and q is denoted by $p.q$. For any term $t \in \mathfrak{T}_{\mathcal{V}}(\Sigma)$, $\mathrm{Pos}(t)$ is the set of all *positions* in t, inductively defined as follows: $\varepsilon \in \mathrm{Pos}(t)$, and if $p \in \mathrm{Pos}(t_i)$ then $i.p \in \mathrm{Pos}(f(t_1, \ldots, t_n))$. The subterm occurring at position p in a term t is denoted by $t|_p$. It is defined inductively by: $t|_\varepsilon \stackrel{\text{def}}{=} t$ and $f(t_1, \ldots, t_n)|_{i.p} \stackrel{\text{def}}{=} t_i|_p$. We denote by $t[s]_p$ the term obtained from t by replacing the term occurring at position p by s, inductively defined as follows: $t[s]_\varepsilon \stackrel{\text{def}}{=} s$ and $f(t_1, \ldots, t_n)[s]_{i.p} \stackrel{\text{def}}{=} f(t_1, \ldots, t_{i-1}, t_i[s]_p, t_{i+1}, \ldots, t_n)$.

Example 2.1 *Let $\Sigma_0 = \{a, b\}$, $\Sigma_1 = \{f\}$, $\Sigma_2 = \{g\}$. The set of terms includes a, b, $f(a)$, $f(f(b))$, $g(f(a), f(b))$, etc. The set of positions in $g(f(a), f(b))$ is $\{\varepsilon, 1, 2, 1.1, 2.1\}$. The term at position* 1.1 *in $g(f(a), f(b)$ is a.*

An *atom* is an expression of the form $s \simeq t$, where $s, t \in \mathfrak{T}_{\mathcal{V}}(\Sigma)$. Atoms are considered modulo commutativity of $\simeq$, i.e. $s \simeq t$ and $t \simeq s$ are viewed as identical. A *literal*, usually denoted by l or m, is either an atom $s \simeq t$ (a *positive literal*, or a literal of *positive polarity*)

2. As usual non-equational predicates $p(t_1, \ldots, t_n)$ are encoded as equations $p(t_1, \ldots, t_n) \simeq \top$.

or the negation of an atom $s \not\simeq t$ (a *negative literal*, or a literal of *negative polarity*). A literal l will sometimes be written $s \bowtie t$, where the symbol $\bowtie$ stands for $\simeq$ or $\not\simeq$. The literal l^{c} denotes the complement of l, i.e., l^{c} is $s \not\simeq t$ (resp. $s \simeq t$) when l is $s \simeq t$ (resp. $s \not\simeq t$).

A *clause* is a finite multiset of literals, usually written as a disjunction. The symbol $\Box$ denotes the empty clause. For every clause C, we define $C^+ \stackrel{\text{def}}{=} \{l \in C \mid l \text{ is positive}\}$ and $C^- \stackrel{\text{def}}{=} \{l \in C \mid l \text{ is negative}\}$. A clause C is *positive* when $C = C^+$, and it is *negative* when $C = C^-$. We denote by $C \backslash D$ the multiset difference of C and D, defined as usual, and we may write $C \backslash l$ instead of $C \backslash \{l\}$. We denote by $|C|$ the length of clause C, i.e. the number of literals it contains. A clause is *unit* if $|C| = 1$. If $C = \bigvee_{i=1}^n l_i$ then $\neg C \stackrel{\text{def}}{=} \bigwedge_{i=1}^n l_i^c$. We often identify sets of unit clauses with conjunctions; for example, given a set of clauses S, we may write $S \cup \bigwedge_{i=1}^n l_i$ instead of $S \cup \{l_i \mid i \in [1,n]\}$, and instead of $\{l_1, \dots, l_n\} \subseteq \{l'_1, \dots, l'_m\}$, we may write $\bigwedge_{i=1}^n l_i \subseteq \bigwedge_{i=1}^m l'_i$.

An expression (term, atom, literal or clause) E is *ground* if it contains no variable. The notation $\mathfrak{T}(\Sigma)$ denotes the set of ground terms and $\mathcal{V}(E)$ denotes the set of variables occurring in E.

Example 2.2 *Consider the clauses $C = f(a) \simeq g(f(a), f(b)) \vee a \not\simeq b$ and $D = g(x,y) \simeq g(y,x) \vee f(x) \simeq f(y)$. Observe that C is ground but D is not (a, b are constant symbols and x, y are variables). We have: $C^+ = f(a) \simeq g(f(a), f(b))$, $C^- = a \not\simeq b$, $D^+ = D$ and $D^- = \Box$.*

A *substitution* is a function mapping variables to terms. The image of a variable x by a substitution σ is denoted by $x\sigma$. Substitutions can be extended to any term, atom, literal, or clause by induction: $a\sigma \stackrel{\text{def}}{=} a$ if $a \in \Sigma_0$, $f(t_1, \dots, t_n)\sigma \stackrel{\text{def}}{=} f(t_1\sigma, \dots, t_n\sigma)$, $(s \simeq t)\sigma \stackrel{\text{def}}{=} (s\sigma \simeq t\sigma)$, $(\bigvee_{i=1}^n l_i)\sigma \stackrel{\text{def}}{=} \bigvee_{i=1}^n l_i\sigma$. The *domain* of a substitution σ is the set of variables x such that $x\sigma \neq x$. A substitution σ is *ground* if $x\sigma$ is ground for every x in the domain of σ. An expression (term, atom, literal or clause) E is an *instance* of an expression E' if there exists a substitution σ such that $E'\sigma = E$. A substitution σ is *more general* than a substitution θ if there exists a substitution η such that $x\sigma\eta = x\theta$, for any variable x. A substitution σ is a *unifier* of two terms t and s if $t\sigma = s\sigma$, in which case t and s are *unifiable*. It is well-known (see for instance Jouannaud & Kirchner, 1991) that any unifiable pair of terms possesses a (unique up to a renaming of variables) most general unifier, denoted by $\mathrm{mgu}(t, s)$.

Example 2.3 *Let $t = f(g(x), g(b))$ and $s = f(g(a), g(y))$. The substitution $\sigma : \{x \mapsto a, y \mapsto b\}$ is the (unique) unifier of t and s.*

We consider a *ground-total strict reduction order* (see, e.g., Baader & Nipkow, 1998) $\prec$ on terms. Such an order satisfies the following properties: $\prec$ is total on ground terms, i.e., if t and s are distinct terms in $\mathfrak{T}(\Sigma)$ then either $t \prec s$ or $s \prec t$; $\prec$ is well-founded, i.e., there is no infinite decreasing sequence $t_1 \succ t_2 \succ \dots \succ t_n \succ \dots$; $\prec$ is closed under substitution, i.e., if $t \prec s$ then $t\sigma \prec s\sigma$; and $\prec$ is closed under contextual embedding, i.e., if $t \prec s$ then $u[t]_p \prec u[s]_p$, for any term u. The order $\prec$ is extended to literals and clauses using the Dershowitz-Manna multiset extension (Dershowitz & Manna, 1979). To this purpose, a negative literal $t \not\simeq s$ is associated with the multiset $\{\{t, s\}\}$ and a positive literal $t \simeq s$ with $\{\{t\}, \{s\}\}$. Examples of reduction orders include Knuth-Bendix orderings or path orderings (see, e.g., Dershowitz, 1979).

Example 2.4 *Let* $C = f(a) \simeq a \vee a \not\simeq b$ *and* $D = g(f(a), f(b)) \simeq a$. *Since* $\prec$ *is well-founded and closed under contextual embedding we necessarily have* $g(f(a), f(b)) \succ f(a) \succ a$ *and* $g(f(a), f(b)) \succ b$*; thus* $D \succ C$.

We now recall some basic notions about rewrite systems (Baader & Nipkow, 1998). A *ground rewrite rule* is a pair written $t \to s$, where t and s are ground terms. A *ground rewrite system* is a set of ground rewrite rules. If R is a ground rewrite system, and u and v are ground terms, we write $u \to_R v$ if there exists a position p such that $u|_p = t$, $v = u[s]_p$, and $t \to s \in R$. The relation $\to_R^*$ denotes the reflexive and transitive closure of $\to_R$.

A term s is a *normal form of* t if $t \to_R^* s$ and there is no term s' such that $s \to_R s'$. A rewrite system is *orthogonal* if there are no rules $t \to s$ and $t' \to s'$ and no position p such that $t|_p = t'$. It is *terminating* if $\to_R$ is well-founded, and *convergent* if every term t admits a unique normal form.

A proof of the following result has been presented by Baader and Nipkow (1998).

Proposition 2.5 *Every orthogonal and terminating rewrite system is convergent.*

2.1.2 Semantics

An *interpretation* is a congruence relation on $\mathfrak{T}(\Sigma)$. The notation $s =_\mathcal{I} t$ is used as a shorthand for $(s, t) \in \mathcal{I}$. An interpretation $\mathcal{I}$ *validates*:

- a ground literal $t \bowtie s$ if $\bowtie$ is $\simeq$ and $s =_\mathcal{I} t$, or $\bowtie$ is $\not\simeq$ and $s \neq_\mathcal{I} t$;
- a ground clause C if $\mathcal{I}$ validates at least one literal in C;
- a non-ground clause C if $\mathcal{I}$ validates all ground instances of C;
- a clause set S if $\mathcal{I}$ validates all clauses in S.

We write $\mathcal{I} \models E$ and say that $\mathcal{I}$ is a *model* of E if the expression (literal, clause or clause set) E is validated by $\mathcal{I}$. 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 of which all interpretations are models and a *contradiction* is a clause that has no model.

2.2 Prime Implicates

We introduce the central notion of a prime implicate, adapted from the corresponding notion in propositional logic (see, e.g., Ngair, 1993).

Definition 2.6 *A ground clause* C *is an* implicate *of a set of clauses* S *if* $S \models C$*; it 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* $C \models D$.

Intuitively, the implicates of S are the clausal consequences of S. They are prime if they are minimal w.r.t. logical entailment and not valid. From a practical point of view, it is clear that implicates that are either valid or not minimal are redundant, hence we are interested only in computing prime implicates.

Example 2.7 *Consider the clause set S:*

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

The clause $d \simeq a$ is an implicate of S, since Clauses 2 and 3 together entail $f(a) \not\simeq f(b)$, which in turn entails $a \not\simeq b$, which together with Clause 1 entail $d \simeq a$. The clause $a \not\simeq e \vee d \simeq e$ can be deduced from Clauses 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 $a \not\simeq e \vee d \simeq e \not\models d \simeq a$.

The following proposition restates the definition of unsatisfiability in terms of implicates.

Proposition 2.8 *A clause set S is unsatisfiable iff $\Box$ is an implicate of S.*

Proof. If S is unsatisfiable then it has no model, thus every clause is an implicate of S. Conversely, if $\mathcal{I} \models S$, and $\Box$ is an implicate of S then $\mathcal{I} \models \Box$, which is impossible, since by definition $\Box$ is false in all interpretations. ∎

Generating all prime implicates is unfeasible in general, since there are infinitely many such clauses. For instance the clause set $\{f(a) \simeq a, f(b) \simeq b, a \not\simeq b\}$ has an infinite number of prime implicates, of the form $f^n(a) \not\simeq f^n(b)$, for every $n \in \mathbb{N}$, and none of them are prime (since $f^{n+1}(a) \not\simeq f^{n+1}(b) \models f^n(a) \not\simeq f^n(n)$. In practice, additional restrictions have to be added to control the form of the generated implicates (for instance to limit the length and depth of the clauses). This yields the following definition:

Definition 2.9 *A ground clause C is a* prime implicate *of a clause set S w.r.t. a class of ground clauses $\mathfrak{C}$ if C is an implicate of S, C is not a tautology, and for every clause $D \in \mathfrak{C}$ such that $S \models D$, we have either $D \not\models C$ or $C \models D$.*

Due to Proposition 2.8, since the satisfiability problem is undecidable in first-order logic, checking whether a given ground clause is an implicate of a set of non-ground clauses S is undecidable. If S is ground, then the problem is co-NP-complete, since it can be polynomialy translated into a propositional entailment problem by flattening the terms and adding all the relevant ground instances of the equality axioms. Similarly, checking whether a set of ground clauses S has an implicate defined over a given set of ground terms is Σ_2^P-complete, since it can be reduced to a $\forall^*\exists^*$-QBF propositional formula (see Eiter & Gottlob, 1995, for more details about the complexity of the abduction problem in propositional logic). In particular, it follows from the results by Eiter and Gottlob (1995) that checking whether a given ground clause is a *prime* implicate of a set of ground clauses is DP-hard.

The following proposition shows that focusing on the computation of ground implicates is actually not restrictive.

Proposition 2.10 *Let S be a set of clauses, C be a clause. Let σ be a substitution mapping all variables in C to pairwise distinct constant symbols not occurring in S or C. Then $S \models C$ holds iff $C\sigma$ is an implicate of S.*

Proof. The direct implication is trivial: if $S \models C$, then necessarily $S \models C\sigma$ (by definition of the semantics of universal quantifiers), thus $C\sigma$ is an implicate of S, by definition of the notion of an implicate. For the other direction, assume that $S \models C\sigma$ and $S \not\models C$. This entails that there exist an interpretation $\mathcal{I}$ and a ground substitution θ of domain $\mathcal{V}(C)$ such that $\mathcal{I} \models S$ and $\mathcal{I} \not\models C\theta$. For every ground expression E (e.g., a literal, a clause…), we denote by E' the expression obtained from E by replacing all occurrences of the constant symbols $x\sigma$ (with $x \in \mathcal{V}(C)$) by $x\theta$. E' is well defined since by hypothesis σ is injective on $\mathcal{V}(C)$. Let $\mathcal{J}$ be the relation defined as follows: $(t, s) \in \mathcal{J}$ iff $(t', s') \in \mathcal{I}$. It is easy to check that $\mathcal{J}$ is a congruence, and that, for every clause D, $\mathcal{J} \models D$ iff $\mathcal{I} \models D'$. Since C contains no constant symbol of the form $x\sigma$, we have $(C\sigma)' = C\theta$; thus $\mathcal{J} \models C\sigma \Leftrightarrow \mathcal{I} \models C\theta$. Therefore $\mathcal{J} \not\models C\sigma$. Since $S \models C\sigma$, this entails that $\mathcal{J} \not\models S$, hence that there exist a substitution η and a clause $D \in S$ such that $\mathcal{J} \not\models D\eta$, i.e., such that $\mathcal{I} \not\models (D\eta)'$. Since S (hence D) contains no constant symbol of the form $x\sigma$, we have $(D\eta)' = D\eta'$, where $x\eta' \stackrel{\text{def}}{=} (x\eta)'$, for every $x \in \mathcal{V}(D)$. This entails that $\mathcal{I} \not\models D\eta'$, which contradicts the fact that $\mathcal{I} \models S$. ∎

Thanks to Proposition 2.10, non-ground implicates can be computed from ground implicates, simply by replacing all constant symbols not occurring in the original clause set by fresh variables[3] (see also Remark 4.16).

3. Handling Ground Clauses Modulo Equality

In this section we focus on ground clauses and provide syntactic characterizations of equivalence and logical entailment between clauses in equational logic. Various notions of redundancy are used in different contexts. For instance complexity results about the computation of minimal representations of CNF formulæ are presented by Liberatore (2005). In the Superposition calculus (Bachmair & Ganzinger, 1994), a clause is considered to be redundant if it is entailed by smaller clauses (w.r.t. the ordering $\prec$), and it is possible to show that such clauses can safely be discarded without threatening refutational completeness. In our case, since we are interested in computing all prime implicates of a formula, an implicate is considered redundant iff it is not prime, i.e., if it is a logical consequence of another implicate. In propositional logic, detecting clauses that are entailed by other clauses is an easy task, because a clause C is a logical consequence of D iff either C is a tautology or every literal in D also occurs in C. It is clear that this can be checked in polynomial time w.r.t. the size of C and D. Furthermore, the only tautologies are the clauses containing complementary literals, which is straightforward to test, and two non-tautological clauses are equivalent iff they are identical (up to AC). In equational logic, these properties do not hold anymore: for example, $a \not\simeq b \vee f(a) \simeq f(b)$ is a tautology, and $e \not\simeq b \vee b \not\simeq c \vee f(a) \simeq f(b)$ is entailed by $e \not\simeq c \vee a \simeq c$. Moreover, a clause may admit several (sometimes exponentially many) equivalent forms, for example, $\bigvee_{i=1}^{n} a_i \not\simeq b_i \vee f(a_1, \dots, a_n) \simeq c$ is equivalent to any clause of the form $\bigvee_{i=1}^{n} a_i \not\simeq b_i \vee f(d_1, \dots, d_n) \simeq c$, where for all $i = 1 \dots, n$, $d_i \in \{a_i, b_i\}$. In this section, we devise a new criterion that generalizes subsumption to test logical entailment, as well as a way to normalize clauses up to equivalence.

3. This requires that terms containing such constants must be allowed as implicates, which of course increases the search space.

3.1 Testing Logical Entailment Between Ground Equational Clauses

We associate each clause C with an equivalence relation $\equiv_C$ on terms. Intuitively, two terms t and s are in relation in $\equiv_C$ iff the equation $t \simeq s$ is a logical consequence of the negation of C. The *C-representative* of a term t is then the smallest (according to the ordering $\prec$) term that is equivalent to t. This notion extends to equations and clauses. Formally:

Definition 3.1 *Given a ground clause C, we define the relation $\equiv_C$ on terms as follows: for all terms $s, t \in \mathfrak{T}(\Sigma)$, $s \equiv_C t$ iff $\neg C \models s \simeq t$. It is clear that $\equiv_C$ is a congruence relation; for any term $s \in \mathfrak{T}(\Sigma)$, the $\equiv_C$-equivalence class of s is called the C-*congruence class *of s and denoted by $[s]_C$.*

*The C-*representatives *of a term s, literal l and clause $l_1 \vee \cdots \vee l_n$ are respectively defined by:*

$$\begin{aligned} s_{\lfloor C} &\stackrel{def}{=} \min_{\prec}([s]_C), \\ (s \bowtie t)_{\lfloor C} &\stackrel{def}{=} s_{\lfloor C} \bowtie t_{\lfloor C}, \\ (l_1 \vee \cdots \vee l_n)_{\lfloor C} &\stackrel{def}{=} l_{1\lfloor C} \vee \cdots \vee l_{n\lfloor C}. \end{aligned}$$

Example 3.2 *Let $C = a \not\simeq b \vee b \not\simeq c$. We have $\neg C = a \simeq b \wedge b \simeq c$, thus by transitivity $\neg C \models a \simeq c$. Consequently, $a \equiv_C b \equiv_C c$. The terms a, b, c are in the same equivalence class. Assuming that $c \prec b \prec a$, we deduce that $a_{\lfloor C} = b_{\lfloor C} = c_{\lfloor C} = c$.*

Example 3.3 *Let $C = a \not\simeq b \vee f(c) \not\simeq d \vee f(b) \simeq f(c)$. Since $\neg C \models a \simeq b$, we also have $\neg C \models f(a) \simeq f(b)$. Moreover $\neg C \models f(c) \simeq d$. Given the order $d \prec c \prec b \prec a \prec f(c) \prec f(b) \prec f(a)$, we have $a_{\lfloor C} = b_{\lfloor C} = b$, $f(a)_{\lfloor C} = f(b)_{\lfloor C} = f(b)$ and $f(c)_{\lfloor C} = d_{\lfloor C} = d$. Thus $(f(b) \simeq f(c))_{\lfloor C} = f(b) \simeq d$.*

The following propositions are straightforward consequences of the previous definition.

Proposition 3.4 *We have $(s \equiv_C t)$ iff $(s \not\simeq t \models C)$ iff $(\neg C \models s \simeq t)$.*

Proof. By definition of $\equiv_C$, $s \equiv_C t$ holds iff $\neg C \models s \simeq t$. By contrapositive this is equivalent to $s \not\simeq t \models C$. ■

Proposition 3.5 *Let s be a term, l be a literal and C and D be two clauses, then:*

$$\neg C \models s \simeq s_{\lfloor C}, \quad \neg C \models l \Leftrightarrow l_{\lfloor C}, \quad \textit{and } \neg C \models D \Leftrightarrow D_{\lfloor C}.$$

Proof. By definition of the notion of a C-representative, we have $s_{\lfloor C} \in [s]_C$, hence $s \equiv_C s_{\lfloor C}$. By definition of $\equiv_C$, this means that $\neg C \models s \simeq s_{\lfloor C}$. The literal l is of the form $s \bowtie t$ with $\bowtie \in \{\simeq, \not\simeq\}$, and by definition $l_{\lfloor C} = (s_{\lfloor C} \bowtie t_{\lfloor C})$. By the previous relation, we have $\neg C \models s \simeq s_{\lfloor C}$ and $\neg C \models t \simeq t_{\lfloor C}$, thus, $\neg C \models (s \bowtie t) \Leftrightarrow (s_{\lfloor C} \bowtie t_{\lfloor C})$, i.e., $\neg C \models l \Leftrightarrow l_{\lfloor C}$. The clause D is of the form $l_1 \vee \cdots \vee l_n$, where $l_1, \ldots, l_n$ are literals. By definition, $D_{\lfloor C} = \bigvee_{i=1}^n l_{i\lfloor C}$. By the previous relation, $\neg C \models l_i \Leftrightarrow l_{i\lfloor C}$ for all i, and $\neg C \models D \Leftrightarrow D_{\lfloor C}$. ■

The next proposition states that it is possible to check that a positive ground clause C is a logical consequence of a set of unit clauses S by testing every equation in C independently. This property does not hold if the clause is not positive, for instance the empty clause set entails the tautology $(a \simeq b \vee a \not\simeq b)$ but entails neither $a \simeq b$ nor $a \not\simeq b$.

Proposition 3.6 *Let $S = \{u_i \simeq v_i \mid i \in \{1 \dots k\}\}$ be a set of unit positive clauses. If $S \models \bigvee_{i=1}^{n}(t_i \simeq s_i)$ where $t_1, \dots, t_n, s_1, \dots, s_n$ are ground terms, then there exists $j \in \{1 \dots n\}$ such that $S \models t_j \simeq s_j$.*

Proof. Assume that $S \models \bigvee_{i=1}^{n}(t_i \simeq s_i)$. Let $\mathcal{I}$ be the interpretation such that $t =_{\mathcal{I}} s$ iff $t =_{\mathcal{I}'} s$ holds for every model $\mathcal{I}'$ of S. Note that $\mathcal{I}$ is a congruence, as it is an intersection of congruences. By definition we have $\mathcal{I} \models u_i \simeq v_i$ for every $i \in \{1 \dots n\}$, thus $\mathcal{I} \models S \models \bigvee_{i=1}^{n}(t_i \simeq s_i)$, and there exists $j \in \{1 \dots n\}$ such that $\mathcal{I} \models t_j \simeq s_j$. This entails that $\mathcal{I}' \models t_j \simeq s_j$ for every model $\mathcal{I}'$ of S, thus $S \models t_j \simeq s_j$. ∎

The next lemma shows that, unless C is a tautology, the relation $\equiv_C$ depends only on the negative literals in C (if C is a tautology then $\neg C$ is unsatisfiable hence $t \equiv_C s$ holds for all terms t and s).

Lemma 3.7 *Let C and D be two non-tautological ground clauses. The following properties hold:*

1. *If $C^- = D^-$ then $\equiv_C$ and $\equiv_D$ are identical; hence $E_{\lfloor C} = E_{\lfloor D}$ holds for every expression E. In particular, $\equiv_C$ and $\equiv_{C^-}$ are identical.*
2. *If $C^- = \square$ then $\equiv_C$ is the identity relation, thus $E_{\lfloor C} = E$ for every expression E.*
3. *The inclusion $\equiv_D \subseteq \equiv_C$ holds iff every negative literal in $D_{\lfloor C}$ is a contradiction.*

Proof. We prove that if C is not a tautology then $\equiv_C$ and $\equiv_{C^-}$ are identical. Note that $\neg C \models \neg C^-$ thus $\equiv_{C^-} \subseteq \equiv_C$, we now show that $\equiv_C \subseteq \equiv_{C^-}$. Let t and s be two terms such that $t \equiv_C s$. Then $\neg C \models t \simeq s$, thus $\neg C^- \wedge \neg C^+ \models t \simeq s$, i.e., $\neg C^- \models C^+ \vee t \simeq s$. By Proposition 3.6 we have either $\neg C^- \models C^+$ or $\neg C^- \models t \simeq s$. In the former case C is a tautology which contradicts our hypothesis. We deduce that $\neg C^- \models t \simeq s$ and $t \equiv_{C^-} s$.

Item 1 follows immediately from the previous property. If $C^- = \square$ then $\equiv_C = \equiv_\square$ and by definition $\equiv_\square$ is the identity relation. If $\equiv_D \subseteq \equiv_C$ then for every negative literal $t \not\simeq s \in D$, we have $t \equiv_D s$ hence $t \equiv_C s$ and $t_{\lfloor C} = s_{\lfloor C}$. Conversely, assume that for every negative literal $t \not\simeq s \in D$, we have $t_{\lfloor C} = s_{\lfloor C}$. Then $\neg C \models t \simeq s$ holds for all $t \not\simeq s \in D$, thus $\neg C \models \neg D^-$, and $\equiv_D \subseteq \equiv_C$. ∎

We now define an adapted notion of subsumption for equational clauses.

Definition 3.8 *Let C, D be two non-tautological ground clauses. The clause D* E-subsumes *C, written $D \leq_{\mathrm{E}} C$, iff the two following conditions hold:*

1. *$\equiv_D \subseteq \equiv_C$,*
2. *for every positive literal $l \in D$, there exists a positive literal $m \in C$ such that $m_{\lfloor C^- \vee l^c}$ is a tautology.*

If S, S' are sets of clauses, we write $S \leq_{\mathrm{E}} C$ if there is a clause $D \in S$ such that $D \leq_{\mathrm{E}} C$, and we write $S \leq_{\mathrm{E}} S'$ if $\forall C \in S', S \leq_{\mathrm{E}} C$.

Intuitively, testing whether $D \models C$ is performed by verifying that $\neg C \models \neg D$. To this purpose, we first check that all equations in $\neg D$ are entailed by $\neg C$, which can be done by verifying that $\equiv_D \subseteq \equiv_C$ holds. Then, we consider the negative literals $t \not\simeq s$ in $\neg D$. We should have $\neg C \models t \not\simeq s$, i.e., $\neg C^- \wedge t \simeq s \models C^+$, which is equivalent by Proposition 3.6 to $\neg C^- \wedge t \simeq s \models m$ for some $m \in C^+$; and this is the case iff $m_{\downharpoonleft C^- \vee l^c}$ where $l = t \simeq s$ is a tautology. We provide an example illustrating both conditions.

Example 3.9 *Let $C = a \not\simeq b \vee b \not\simeq c \vee f(c) \simeq f(d)$, $D = a \not\simeq c \vee c \simeq d$, and assume $d \prec c \prec b \prec a$. By definition, we have $a \equiv_C b \equiv_C c$, hence $\equiv_D \subseteq \equiv_C$. Consider the literal $l = c \simeq d$ in D and let $m = f(c) \simeq f(d)$. We have $C^- \vee l^c = a \not\simeq b \vee b \not\simeq c \vee c \not\simeq d$, hence a, b, c, d are all equivalent w.r.t. $\equiv_{C^- \vee l^c}$. Thus $m_{\downharpoonleft C^- \vee l^c} = f(d) \simeq f(d)$ is a tautology, and D E-subsumes C.*

Theorem 3.10 *Let C and D be two non-tautological ground clauses. We have $D \models C$ iff $D \leq_{\mathrm{E}} C$.*

Proof. First assume that $D \models C$. Consider a negative literal $l = s \not\simeq t$ in D. We have $\neg C \models \neg D \models l^c$, thus $t_{\downharpoonleft C} = s_{\downharpoonleft C}$ by Proposition 3.4, and $l_{\downharpoonleft C}$ is a contradiction. By Lemma 3.7 (3), we deduce that $\equiv_D \subseteq \equiv_C$. Now consider a positive literal $l \in D$. By hypothesis $\neg C \models l^c$, thus $\neg C^- \wedge \neg C^+ \models l^c$ and $\neg C^- \wedge l \models C^+$. By Proposition 3.6 we deduce that there exists $m \in C^+$ such that $\neg C^- \wedge l \models m$, i.e., by Definition 3.1, $m_{\downharpoonleft C^- \vee l^c}$ is a tautology.

For the converse implication, we prove that every literal in D entails C. If $s \not\simeq t \in D$, then since $s \equiv_D t$, we deduce that $s \equiv_C t$ and by Proposition 3.4, $s \not\simeq t \models C$. If $s \simeq t \in D$, then by hypothesis there is a literal $u \simeq v \in C$ such that $u_{\downharpoonleft C^- \vee s \not\simeq t} = v_{\downharpoonleft C^- \vee s \not\simeq t}$. It follows that $\neg(C^- \vee s \not\simeq t) \models u \simeq v$, hence $s \simeq t \models (C^- \vee u \simeq v) \models C$. ∎

Example 3.11 *Given the order $a \prec b \prec c \prec e \prec f(a) \prec f(b)$ on terms, let $C = e \not\simeq b \vee b \not\simeq c \vee f(a) \simeq f(b)$, $D = e \not\simeq c \vee a \simeq c$, and let $l = e \not\simeq c$ and $m = a \simeq c$ be the literals of D. We have $l_{\downharpoonleft C} = b \not\simeq b$ because $[b]_C = \{b, c, e\}$ and $\min_{\prec}([b]_C) = b$. Moreover the literal $f(a) \simeq f(b) \in C$ is such that $(f(a) \simeq f(b))_{\downharpoonleft C \vee m^c} = f(a) \simeq f(a)$, which, by Theorem 3.10, proves that $D \models C$.*

3.2 A Normal Form for Ground Equational Clauses

The following definition introduces the notion of a *normalized clause*, which in particular permits to efficiently test whether a clause is tautological and whether two clauses are equivalent. The intuition underlying this definition is that the relation $\equiv_C$ can be defined in a canonical way by stating that each term t is mapped to its normal form $t_{\downharpoonleft C}$, which is expressed by the negative literal $t \not\simeq t_{\downharpoonleft C}$ when $t \neq t_{\downharpoonleft C}$. Afterwards, each positive literal can be replaced by its normal form, and the literals that are redundant w.r.t. substitutivity can be removed. For example, the literal $a \simeq b$ is redundant in $f(a) \simeq f(b) \vee a \simeq b$ because $f(a) \simeq f(b) \equiv a \simeq b \vee f(a) \simeq f(b)$.

Definition 3.12 *A ground clause C is* normalized *if:*

1. *every literal l in C is such that $l_{\downarrow C \setminus l} = l$;*

2. *there are no two distinct positive literals l, m in C such that $m_{\downarrow l^c \vee C^-}$ is a tautology;*

3. *C contains no literal of the form $t \not\simeq t$.*

Example 3.13 *Using the ordering $a \prec b \prec c \prec e \prec f(a) \prec f(b)$ on terms, the clause $C = c \not\simeq b \vee e \not\simeq b \vee f(b) \simeq f(a)$ is normalized.*

Proposition 3.14 *If C is a normalized clause, then each literal in C occur exactly once in C.*

Proof. This follows immediately from Item 1 of Definition 3.12 for negative literals and Item 2 for positive literals. ∎

Proposition 3.15 *A normalized clause C is a tautology iff C contains a tautological literal.*

Proof. If C is a tautology, then $\neg C^- \models C^+$, hence by Proposition 3.6 there exists a literal l of the form $s \simeq t$ in C^+ such that $\neg C^- \models l$. By Proposition 3.4 we have $s_{\downarrow C} = t_{\downarrow C}$, and since C is normalized, we deduce that $l_{\downarrow C} = l_{\downarrow C \setminus l} = l$, and that the latter is a tautological literal. ∎

The proof that non-tautological normalized clauses that are equivalent are identical relies on a ground rewriting system that is associated with normalizing clauses.

Definition 3.16 *Let C be a normalized clause. The rewriting system R_C associated with C is defined by:*

$$R_C \stackrel{\text{def}}{=} \{t \to s \mid t \not\simeq s \in C \wedge s \prec t\}.$$

The rules in R_C are ordered according to the lexicographic extension of $\prec$.

By construction, since C is normalized, R_C is orthogonal. Furthermore, if $t \to_{R_C} s$ then $t \succ s$, which entails that R_C is terminating, hence convergent by Proposition 2.5. We denote by $t_{\downarrow R_C}$ the R_C-normal form of t; note that $t_{\downarrow R_C} = t_{\downarrow C}$.

Proposition 3.17 *Let C be a normalized clause and t be a term in $\mathfrak{T}(\Sigma)$. The term $t_{\downarrow R_C}$ is obtained from t by using only rules of R_C that have a left-hand side smaller or equal to t.*

Proof. To compute $t_{\downarrow R_C}$, rewriting rules can only be applied on t or its subterms. All the subterms of t are smaller than t by $\prec$ and by definition of R_C, a rewriting step always replaces a term by a smaller one. ∎

Theorem 3.18 *Two non-tautological equivalent and normalized clauses are identical. Furthermore, for any non-tautological ground clause C, there exists a normalized clause equivalent to C, and this clause is the smallest ground clause equivalent to C.*

Proof. Consider two non-tautological, equivalent and normalized clauses C_1 and C_2. Since C_1 and C_2 are equivalent, $\equiv_{C_1}$ and $\equiv_{C_2}$ are identical.

We first show that ${C_1}^- = {C_2}^-$. We denote by R_1 and R_2 the rewriting systems associated with C_1 and C_2 respectively. If ${C_1}^- \neq {C_2}^-$, then $R_1 \neq R_2$. We consider the smallest rule $t \to s$ that does not appear in both rewriting systems; w.l.o.g. we assume that $t \to s \in R_1$ and $t \to s \notin R_2$, hence that $t \not\simeq s \in C_1$. Since C_1 and C_2 are equivalent and $t_{\lfloor R_1} = s$, we also have $t_{\lfloor R_2} = s$. By Proposition 3.17, the left-hand sides of the rules used to rewrite t are smaller or equal to t, and since t is eventually rewritten into s in R_2, there must be at least one rule $u \to v$ in R_2 that can be applied to t; t is thus of the form $t[u]$. If u is a strict subterm of t, then $u \to v$ is smaller than $t \to s$ and by hypothesis, $u \to v$ occurs in R_1, so that $u \not\simeq v \in C_1$. But in this case, $(t \not\simeq s)_{\lfloor C_1 \setminus t \not\simeq s}$ and $t \not\simeq s$ are distinct, which contradicts Point 1 of Definition 3.12. Otherwise, we must have $u = t$ and $t \succ v \succ s$. But then $t_{\lfloor R_2} = v \neq s$, a contradiction.

We now show that ${C_1}^+ = {C_2}^+$. By Theorem 3.10 we have $C_1 \leq_{\mathrm{E}} C_2$ and $C_2 \leq_{\mathrm{E}} C_1$. Let l_1 be a positive literal in C_1. Since $C_1 \leq_{\mathrm{E}} C_2$, there exists a literal l_2 in C_2 such that ${l_2}_{\lfloor {l_1}^c \vee {C_2}^-}$ is a tautology, i.e., $l_1 \wedge \neg {C_2}^- \models l_2$; hence $l_1 \models {C_2}^- \vee l_2$. Similarly, since $C_2 \leq_{\mathrm{E}} C_1$ and $l_2 \in C_2$, there is a positive literal m_1 in C_1 such that ${m_1}_{\lfloor {l_2}^c \vee {C_1}^-}$ is a tautology, so that $l_2 \models {C_1}^- \vee m_1$. We deduce that

$$l_1 \;\models\; {C_2}^- \vee l_2 \;\models\; {C_2}^- \vee {C_1}^- \vee m_1 \;\equiv\; {C_1}^- \vee m_1,$$

and therefore, $\neg {C_1}^- \wedge l_1 \models m_1$ and ${m_1}_{\lfloor {C_1}^- \vee {l_1}^c}$ is a tautology. By Point 2 of Definition 3.12, we must have $m_1 = l_1$. From $l_1 \models {C_2}^- \vee l_2$ and $l_2 \models {C_1}^- \vee l_1$, we deduce that ${l_1}_{\lfloor C_1} \models {l_2}_{\lfloor C_1}$ and ${l_2}_{\lfloor C_2} \models {l_1}_{\lfloor C_2}$, i.e., ${l_1}_{\lfloor C_1} \equiv {l_2}_{\lfloor C_2}$ (because ${C_1}^- = {C_2}^-$) and so either ${l_1}_{\lfloor C_1}$ and ${l_2}_{\lfloor C_2}$ are both tautological or ${l_1}_{\lfloor C_1} = {l_2}_{\lfloor C_2}$. But C_1 and C_2 are non-tautological, thus necessarily $l_1 = l_2$. We conclude that ${C_1}^+ = {C_2}^+$.

Now, consider a non-tautological ground clause C that is not normalized. We show that there exists a ground clause equivalent to C and strictly smaller than C (then the proof follows by an immediate induction).

- If C contains two positive literals l and m such that $m_{l^c \vee C^-}$ is a tautology, then removing the literal l yields a smaller equivalent clause.
- If C contains a literal l such that $l_{\lfloor C \setminus l} \neq l$ then replacing this literal with $l_{\lfloor C \setminus l}$ yields a smaller equivalent clause.
- Removing literals of the form $t \not\simeq t$ yields a smaller equivalent clause. ∎

Definition 3.19 *For every non tautological ground clause, we denote by $C_\downarrow$ the (unique) normalized clause equivalent to C.*

Example 3.20 *The clause $C = c \not\simeq b \vee e \not\simeq b \vee f(b) \simeq f(a)$ is normalized and equivalent to the clauses $c \not\simeq b \vee e \not\simeq b \vee f(c) \simeq f(a)$, $c \not\simeq b \vee e \not\simeq b \vee f(e) \simeq f(a)$, $c \not\simeq e \vee e \not\simeq b \vee f(b) \simeq f(a)$, etc.*

4. An Abductive Superposition Calculus

We now present the calculus for generating implicates, named c$\mathcal{SP}$. This calculus is based on the Superposition Calculus $\mathcal{SP}$ (see Bachmair & Ganzinger, 1994), which is the most efficient proof procedure for first-order logic with equality (see for instance Nieuwenhuis & Rubio, 2001, or `http://www.cs.miami.edu/~tptp/CASC/`) and forms the basis of the most powerful theorem provers currently available (McCune, 2010; Schulz, 2013; Voronkov, 1995; Weidenbach, Afshordel, Brahm, Cohrs, Engel, Keen, Theobalt, & Topic, 2001). The calculus can be seen as a refinement of the well-known Resolution calculus (Robinson, 1965), tuned to handle equations efficiently. The key inference rule is the Superposition rule, which replaces equals by equals, and for efficiency, equations are oriented in such a way that a term may be replaced only by smaller terms. The principle underlying c$\mathcal{SP}$ is to apply the inference rules of $\mathcal{SP}$ to the set of clauses under consideration along with ground unit clauses that are added during proof search and that act as hypotheses. The hypotheses that permit the generation of the empty clause are extracted: they represent the negation of an implicate. In order to keep track of the hypotheses that were used to derive the empty clause, they are attached to standard clauses as constraints. Thus, c$\mathcal{SP}$ is an adaptation of $\mathcal{SP}$ to so-called *constrained clauses*, along with inference rules that permit to generate additional hypotheses.

Before introducing the formal definition of c$\mathcal{SP}$, we provide an example that should give an intuitive understanding of this calculus.

Example 4.1 *Consider the clause set: $S = \{f(a) \simeq c, f(b) \simeq d, c \not\simeq d\}$. It easy to check that $a \not\simeq b$ is an implicate of S. Indeed, by contradiction, for every interpretation $\mathcal{I}$ such that $\mathcal{I} \models S$ and $\mathcal{I} \models a \simeq b$, we have $f(a) =_{\mathcal{I}} f(b)$, thus by the first two clauses we deduce that $c =_{\mathcal{I}} d$, which contradicts the third clause. Consequently, $S \models a \not\simeq b$. We now show how this clause can be derived from S using the principles underlying c$\mathcal{SP}$, assuming $d \prec c \prec b \prec a$. We first remark that, since $b \prec a$, a can be rewritten to b in the clause $f(a) \simeq c$, provided the equation $a \simeq b$ is added as a new hypothesis. This is asserted by deriving the constrained clause $[f(b) \simeq c \mid a \simeq b]$, which is interpreted as an implication: $a \simeq b \Rightarrow f(b) \simeq c$. Afterwards, the clause $[d \simeq c \mid a \simeq b]$ can be derived by replacing $f(b)$ by d, using the second clause in S. We may use the equality $d \simeq c$ to replace c by d in the third clause in S, yielding $[d \not\simeq d \mid a \simeq b]$. Note that the constraint $a \simeq b$ is propagated from the premise to the conclusion of the rule. Finally, the contradictory literal $d \not\simeq d$ can be deleted, yielding the constrained clause $[\square \mid a \simeq b]$. This clause means that the empty clause is derivable from S if $a \simeq b$ is added as an hypothesis. By soundness, this implies that $S \cup \{a \simeq b\}$ is unsatisfiable, hence that $a \not\simeq b$ is an implicate of S.*

All the rules applied in the previous example are the usual inference rules of $\mathcal{SP}$ (adapted to constrained clauses), except the first one which was used to assert a new hypothesis in the constraint part of the clause – the corresponding rule is called *Assertion*. In this case, the asserted hypothesis was positive: an equation $a \simeq b$ was added as a hypothesis so that a could be rewritten into b. We provide another example, showing that the assertion rule may be applied also on negative literals.

Example 4.2 *Consider the clause set: $S = \{a \simeq b, f(b) \simeq c\}$. It is clear that $f(a) \simeq c$ is an implicate of S (it suffices to replace b by a in the second clause). However, assuming that*

$c \prec b \prec a$, this clause cannot be derived by the rules of the Superposition calculus because a is strictly greater than b. The implicate is generated as follows. By adding the hypothesis $f(a) \not\simeq c$ to the clause $a \simeq b$, we may derive $[f(b) \not\simeq c \,|\, f(a) \not\simeq c]$, meaning that $f(b) \not\simeq c$ can be derived from $f(a) \not\simeq c$ and S. Note that this time, the replacement is compatible with the ordering on terms. Afterwards, the term $f(b)$ can be replaced by c by using the second clause in S, yielding a contradiction. We thus obtain the constrained clause $[\Box \,|\, f(a) \not\simeq c]$, meaning that $f(a) \simeq c$ is indeed an implicate of S.

4.1 Constrained Clauses

We begin by defining the syntax and semantics of constrained clauses.

Definition 4.3 *A* constraint *is a set of ground literals, sometimes written as a conjunction. The empty constraint is denoted by $\top$. If $\mathcal{X} = \{l_1, \dots, l_n\}$ then $\mathcal{X}^{\mathrm{c}}$ denotes the clause $\bigvee_{i=1}^{n} l_i^c$. We denote by $\mathcal{X}_{\downarrow}$ the constraint $\neg(\mathcal{X}^{\mathrm{c}}{}_{\downarrow})$, and we say that $\mathcal{X}$ is* normalized *if $\mathcal{X}_{\downarrow} = \mathcal{X}$.*

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

Example 4.4 *The expression $[f(a) \simeq b \,|\, a \simeq b, b \simeq c]$ is a constrained clause. If $c \prec b \prec a$ then $(a \not\simeq b \vee b \not\simeq c)_{\downarrow} = a \not\simeq c \vee b \not\simeq c$, hence $\{a \simeq b, b \simeq c\}_{\downarrow} = \{a \simeq c, b \simeq c\}$.*

From a semantic point of view, a constrained clause $[C \,|\, \mathcal{X}]$ is equivalent to the standard clause $\mathcal{X}^c \vee C$. More specifically, the intended meaning of a c-clause $[C \mid \mathcal{X}]$ is that the clause C can be inferred provided the literals in $\mathcal{X}$ are added as axioms to the considered clause set.

Remark 4.5 *Note that, as stated in Definition 4.3, the constraint part of a c-clause is always ground. This is sufficient in our context: since implicates are universally quantified formulæ, their negation is necessarily ground (after Skolemization). See also Proposition 2.10 and the following example.*

Example 4.6 *Consider the clause set: $S = \{p(x, y), \neg p(x, y) \vee q(x, y)\}$. It is clear that the (non-ground) clause $q(x, y)$ is an implicate of S. The negation of C is $\neg\forall x, y\, q(x, y) \equiv \exists x, y\, \neg q(x, y)$ (since variables are implicitly universally quantified in clauses), and its Skolemized normal form is $\neg q(a, b)$, where a, b are new Skolem constants. The calculus c$\mathcal{SP}$ generates the constrained clause $[\Box \mid \neg q(a, b)]$, which can be transformed afterwards into the implicate $q(a, b)$ or, because a, b do not occur in S, $\forall x, y\, q(x, y)$.*

4.2 Inference Rules

The calculus c$\mathcal{SP}$ is defined by a set of inference rules that are represented in Figure 1. The calculus is parameterized by the ordering $\succ$ on terms and by a selection function *sel* mapping every clause C to a subset of C, where $sel(C)$ contains either all maximal literals in C or at least one negative literal. A literal is *selected* in C if it occurs in $sel(C)$.

The calculus is also parameterized by a set of constraints $\mathfrak{X}$ satisfying the following conditions:

1. $\mathfrak{X}$ is not empty.
2. $\mathfrak{X}$ is $\subseteq$-*closed*, i.e., for every $\mathcal{X} \in \mathfrak{X}$ and constraint $\mathcal{Y}$, if $\mathcal{Y} \subseteq \mathcal{X}$ then $\mathcal{Y} \in \mathfrak{X}$.
3. Every constraint in $\mathfrak{X}$ is normalized.

Note that Conditions 1 and 2 together imply that $\top \in \mathfrak{X}$. Condition 3 is not restrictive since any constraint is equivalent to a normalized constraint. Moreover, if $\mathcal{X}$ is normalized then any subset of $\mathcal{X}$ is also normalized.

Intuitively, the set $\mathfrak{X}$ allows a user to control the form of the candidate implicates. For instance the user may be interested only in positive implicates, or in implicates of small size (i.e., of a size up to some fixed $k \in \mathbb{N}$). Instead of filtering the implicates a posteriori, it is more efficient to block the generation of clauses with constraints that do not fulfill the required property. The condition on the implicates can also be of a semantic nature, for example, the user may want to obtain all implicates that entail some formula (then $\mathfrak{X}$ is the set of constraints that are logical consequences of the negation of the considered formula). All these conditions fulfill the above restrictions. It is also clear that filters can be freely combined, i.e., if $\mathfrak{X}_1$ and $\mathfrak{X}_2$ fulfill Conditions 1-3 above, then so do $\mathfrak{X}_1 \cap \mathfrak{X}_2$ and $\mathfrak{X}_1 \cup \mathfrak{X}_2$.

The c-Superposition, c-Factoring and c-Reflection rules are straightforward adaptations of the usual rules of $\mathcal{SP}$ to c-clauses: the constraints of the premises are added to the conclusion. The Positive and Negative Assertion rules are the ones that permit the addition of new hypotheses as constraints to c-clauses.

The standard Superposition calculus $\mathcal{SP}$ can be recovered from c$\mathcal{SP}$ by setting $\mathfrak{X}$ to $\{\top\}$, in which case the Assertion rules never apply and all constraints are $\top$.

We provide examples of applications of each rule.

Example 4.7 *Consider the c-clauses:*

$$\begin{array}{cc}
1 & [f(a,x) \simeq c \,|\, a \simeq b] \\
2 & [f(y,b) \simeq d \,|\, a \simeq b] \\
3 & [f(y,b) \simeq d \,|\, b \simeq c] \\
4 & [f(a,x) \not\simeq f(y,b) \vee x \not\simeq b \,|\, a \simeq b] \\
5 & [f(a,x) \simeq b \vee f(y,b) \simeq b \,|\, \top]
\end{array}$$

We assume that $d \prec c \prec b \prec a$, that sel$(C)$ is the set of maximal literals in C and that $\mathfrak{X}$ is the entire set of normalized constraints.

The c-Superposition rule applies on c-clauses 1 and 2, with unifier $\sigma = \{x \mapsto b, y \mapsto a\}$ (since we have $f(a,b) \succ a, b, c, d$) and the c-clause $[d \simeq c \,|\, a \simeq b]$ is derived. In contrast, the rule does not apply on c-clauses 1 and 3. Indeed, the constraint $\{a \simeq b, b \simeq c\}$ obtained by taking the union of the constraints of 1 and 3 is not normalized hence does not occur in $\mathfrak{X}$.

The c-Reflection rule applies on c-clause 4, yielding: $[b \not\simeq b \mid a \simeq b]$. Afterwards the rule may be applied again on the literal $b \not\simeq b$, yielding $[\Box \mid a \simeq b]$. Note that the rule does not apply on the literal $x \not\simeq b$ of the initial clause 4 because it is not maximal (since $f(a,x) \succ x, b$).

The c-Factorization rule applies on c-clause 5, yielding: $[f(a,b) \simeq b \vee b \not\simeq b \,|\, \top]$.

The Positive Assertion rule applies on c-clause 1. Taking $u = f(a,x)$, $t = t' = a$ and $s = b$, we get $[f(b,x) \simeq c \,|\, a \simeq b]$. Note that we cannot apply the rule with a term s distinct *from b because the obtained constraint* $\{a \simeq b, a \simeq s\}$ *would not be normalized. For the same reason the rule cannot be applied on the term* $f(a,x)$*. It cannot be applied on* x *because this term is a variable, or on* c *because the term is not maximal.*

The Positive Assertion rule applies in several ways on c-clause 5*, for instance, taking* $u = f(a,x)$*,* $t = t' = a$ *and* $t = c$*, we get:* $[f(c,x) \simeq b \vee f(y,b) \simeq b \,|\, a \simeq c]$*.*

The Negative Assertion rule also applies on c-clause 5*. For instance, taking* $u = g(f(a,b))$*,* $v = d$*,* $t = f(a,x)$*,* $t' = f(a,b)$*, and* $s = b$*, we obtain* $[g(b) \not\simeq d \vee f(y,b) \simeq b \,|\, g(f(a,b)) \not\simeq d]$*.*

Remark 4.8 *The term* s' *in the Positive Assertion and the term* u *in the Negative Assertion rules are arbitrary. There are infinitely many terms that can be used with these inference rules. Let us consider for example the c-clause* $[b \simeq a \,|\, \mathcal{X}]$*, defined over a signature containing the unary function symbol* f*. In this case, any term of the form* $f^n(b)$ *for* $n \in \mathbb{N}$ *can all be introduced in the Negative Assertion rule. This is why in practice it is critical to limit the amout of usable terms by imposing restictions on the set* $\mathfrak{X}$

4.3 Redundancy Criterion

An important feature of the Superposition calculus is the availability of a general criterion for detecting redundant clauses. In this context, clauses are considered to be redundant if they can be safely discarded without threatening refutational completeness. In the present section, we adapt this criterion to c$\mathcal{SP}$. The definition is similar to that by Bachmair and Ganzinger (1994), except that the constraints must be taken into account. In $\mathcal{SP}$, a clause is redundant if it is entailed by smaller clauses. In c$\mathcal{SP}$, it is also required that the constraints of the entailing clauses are included in the constraint of the considered clause. Furthermore, the literals occurring in this constraint can also be used in the entailment test, provided they are smaller than the considered clause. Formally:

Definition 4.9 *If* $\mathcal{X}$ *is a constraint and* C *is a clause, we denote by* $\mathcal{X}|_{\preceq C}$ *the set of literals in* $\mathcal{X}$ *that are smaller than or equal to* C*.*

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

- $\forall i \in \{1 \ldots n\}$*,* $C\sigma \succeq D_i\theta_i$ *and* $\mathcal{Y}_i \subseteq \mathcal{X}$*,*
- $\mathcal{X}|_{\preceq C\sigma}, D_1\theta_1, \ldots, D_n\theta_n \models C\sigma$*.*

Example 4.10 *Assume that* $d \prec c \prec b \prec a$*. The c-clause* $[f(a,b) \simeq c \vee a \not\simeq b \vee a \not\simeq c \,|\, c \simeq d \wedge a \simeq b]$ *is redundant w.r.t.* $S = \{[a \not\simeq b \vee b \not\simeq c \,|\, c \simeq d]\}$*. Indeed, it is clear that we have* $a \not\simeq b \vee b \not\simeq c \models f(a,b) \simeq c \vee a \not\simeq b \vee a \not\simeq c$*,* $a \not\simeq b \vee b \not\simeq c \prec f(a,b) \simeq c \vee a \not\simeq b \vee a \not\simeq c$*, and* $\{c \simeq d\} \subseteq \{c \simeq d, a \simeq b\}$*. However, the c-clause* $[a \not\simeq c \vee b \not\simeq c | c \simeq d]$ *is not redundant w.r.t.* S*, although* $a \not\simeq b \vee b \not\simeq c \models a \not\simeq c \vee b \not\simeq c$*, because* $a \not\simeq b \vee b \not\simeq c \succ a \not\simeq c \vee b \not\simeq c$*. Similarly,* $[a \not\simeq b \vee b \not\simeq c | \top]$ *is not redundant w.r.t.* S*, because* $\{c \simeq d\} \not\subseteq \top$*.*

The c-clause $[a \simeq b \,|\, a \simeq c \wedge b \simeq c]$ *is redundant w.r.t.* $\{[a \simeq d \,|\, a \simeq c], [b \simeq d \,|\, b \simeq c]\}$*, because* $a \simeq d, b \simeq d \models a \simeq b$*;* $a \simeq d, b \simeq d \prec a \simeq b$ *and* $\{a \simeq c\} \cup \{b \simeq c\} \subseteq \{a \simeq c, b \simeq c\}$*.*

The c-clause $[f(a) \simeq f(b) \,|\, a \simeq b]$ *is redundant w.r.t. the empty c-clause* $[\square \,|\, \top]$*, because* $a \simeq b|_{\preceq f(a) \simeq f(b)} = (a \simeq b) \models (f(a) \simeq f(b))$*.*

c-Superposition	$\dfrac{[t \simeq s \vee C \,\vert\, \mathcal{X}] \quad [u[t'] \bowtie v \vee D \,\vert\, \mathcal{Y}]}{[u[s] \bowtie v \vee C \vee D \,\vert\, \mathcal{X} \cup \mathcal{Y}]\sigma}$	$(i), (ii), (iii),$
c-Factoring	$\dfrac{[t \simeq u \vee t' \simeq v \vee C \,\vert\, \mathcal{X}]}{[t \simeq u \vee u \not\simeq v \vee C \,\vert\, \mathcal{X}]\sigma}$	$(iv), (ix)$
c-Reflection	$\dfrac{[t \not\simeq t' \vee C \,\vert\, \mathcal{X}]}{[C \,\vert\, \mathcal{X}]\sigma}$	$(v), (ix)$
Positive Assertion	$\dfrac{[u[t] \bowtie v \vee C \,\vert\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,\vert\, \mathcal{X} \wedge t' \simeq s]\sigma}$	$(vi), (vii)$
Negative Assertion	$\dfrac{[t \simeq s \vee C \,\vert\, \mathcal{X}]}{[u[s] \bowtie v \vee C \,\vert\, \mathcal{X} \wedge u[t'] \bowtie v]\sigma}$	$(ii), (viii), (x)$

where the following conditions hold:

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

(i): $(u[t'] \bowtie v)\sigma \in sel((u[t'] \bowtie v \vee D)\sigma)$ and $v\sigma \not\succ u[t']\sigma$;

(ii): $(t \simeq s)\sigma \in sel((t \simeq s \vee C)\sigma)$ and $s\sigma \not\succ t\sigma$;

(iii): $\mathcal{X} \cup \mathcal{Y} \in \mathfrak{X}$;

(iv): $(t \simeq u)\sigma \in sel((C \vee t \simeq u \vee t' \simeq v)\sigma)$;

(v): $(t \simeq t')\sigma \in sel((t \simeq t' \vee C)\sigma)$;

(vi): $s \prec t'$, $v\sigma \not\succ u[t]\sigma$ and $(u[t] \bowtie v)\sigma \in sel((u[t] \bowtie v \vee C)\sigma)$;

(vii): $\mathcal{X} \wedge t' \simeq s \in \mathfrak{X}$;

(viii): $v \prec u[t']$;

(ix): $\mathcal{X} \in \mathfrak{X}$;

(x): $\mathcal{X} \wedge u[t'] \bowtie v \in \mathfrak{X}$.

Figure 1: Inference Rules for c$\mathcal{SP}$.

Remark 4.11 *Note that constraints are compared using set inclusion and not logical entailment. For instance the c-clause $[a \simeq b \,\vert\, f(c) \simeq f(d)]$ is* not *redundant w.r.t. the set $\{[a \simeq b \,\vert\, c \simeq d]\}$, although $c \simeq d \models f(c) \simeq f(d)$. Using logical entailment instead of set inclusion would make the redundancy criterion more general (in the sense that more c-clauses would be detected as redundant), unfortunately it would also make the calculus incomplete. More precisely, this relaxed notion of redundancy is not compatible with the assumption that the constraints in $\mathfrak{X}$ are normalized. Experiments show that this assumption makes the calculus more efficient, even with a more restrictive version of the redundancy elimination rule.*

Proposition 4.12 *Let C be a ground clause. If $C_{\downarrow}$ is redundant w.r.t. S, then C is redundant w.r.t. S.*

Proof. This follows immediately from the fact that $C \succeq C_{\downarrow}$ and $C_{\downarrow} \models C$. ■

4.4 Soundness and Completeness of c$\mathcal{SP}$

In this section, we establish the soundness and completeness of c$\mathcal{SP}$. More precisely, we prove that a clause C is a prime implicate of S iff the c-clause $[\Box \,|\, \neg C]$ can be derived from S using the rules in c$\mathcal{SP}$. The redundancy criterion of Section 4.3 is taken into account. We thus introduce the notion of a saturated set, and we show that saturated sets obtained from S necessarily contain all prime implicates of S.

Definition 4.13 *A set of c-clauses S is* c$\mathcal{SP}$-saturated *if every c-clause deducible by the rules of c$\mathcal{SP}$ (in one step) is redundant w.r.t. S. A* c$\mathcal{SP}$-saturation *of a set of c-clauses S is a set of c-clauses S^* such that: (i) every c-clause in S is redundant w.r.t. S^*, (ii) every c-clause in S^* is obtained from those in S by a finite number of applications of the rules in c$\mathcal{SP}$, (iii) S^* is c$\mathcal{SP}$-saturated.*

Lemma 4.14 *Let $[C \,|\, \mathcal{X}]$ be a c-clause derived (in one step) from premises $[D_i \,|\, \mathcal{Y}_i]$, where $i \in \{1 \dots n\}$. Then $\mathcal{Y}_i \subseteq \mathcal{X}$ for all $i \in \{1 \dots n\}$, $\mathcal{X} \subseteq \mathfrak{X}$ and $\{D_1, \dots, D_n, \mathcal{X}\} \models C$.*

Proof. It is easy to verify that this property holds for each inference rule (using the fact that, by the application conditions (*iii*), (*vii*), (*ix*), (*x*), in Figure 1, an inference is allowed only if the constraint of the conclusion is in $\mathfrak{X}$). ■

Lemma 4.14 permits to deduce the following soundness result:

Lemma 4.15 *Let S be a set of standard clauses and let S^* be a c$\mathcal{SP}$-saturation of S. For every c-clause $[C \,|\, \mathcal{X}] \in S^*$, we have $S \cup \mathcal{X} \models C$ and $\mathcal{X} \in \mathfrak{X}$. In particular, if $C = \Box$, then $S \models \neg\mathcal{X}$, i.e., $\neg\mathcal{X}$ is an implicate of S.*

Proof. The proof is a straightforward induction on the length of the derivation. ■

Remark 4.16 *Recall that if C is an implicate of S, then, as shown in Proposition 2.10, the non-ground clause D obtained from C by replacing each constant not occurring in S by a new variable is also an implicate of S. This remark means that it is possible to use c$\mathcal{SP}$ to compute non-ground implicates: it is sufficient for $\mathfrak{X}$ to contain the Skolemized form of the corresponding candidates (i.e., $\mathfrak{X}$ should contain constraints with constants not occurring in S).*

We now prove that c$\mathcal{SP}$ is deductive-complete, i.e., that it permits to generate every prime implicate of a given set of clauses. The proof relies on the following definition and proposition.

Definition 4.17 *For every set of c-clauses S and for every constraint $\mathcal{X}$, we denote by $S|_{\mathcal{X}}$ the set of standard clauses D such that $[D \,|\, \mathcal{Y}] \in S$ and $\mathcal{Y} \subseteq \mathcal{X}$.*

Proposition 4.18 *Let S be a set of c-clauses and $\mathcal{X}$ be a satisfiable constraint. If a c-clause $[C \mid \mathcal{Y}]$ is redundant w.r.t. S and $\mathcal{Y} \subseteq \mathcal{X}$, then C is redundant w.r.t. $S|_{\mathcal{X}} \cup \mathcal{X}$.*

Proof. By definition of c-clause redundancy (Definition 4.9), there are two cases to consider.

- If $\mathcal{Y}$ is unsatisfiable, then since $\mathcal{Y}$ is a conjunction of literals and $\mathcal{Y} \subseteq \mathcal{X}$, the constraint $\mathcal{X}$ is also unsatisfiable which contradicts the hypothesis of the lemma.
- Otherwise, for every ground substitution σ, there exist c-clauses $[D_i \mid \mathcal{Y}_i] \in S$ ($1 \leq i \leq n$) and ground substitutions θ_i ($1 \leq i \leq n$) such that $\forall i \in \{1 \dots n\}$, $D_i\theta_i \preceq C\sigma$, $\mathcal{Y}_1, \dots, \mathcal{Y}_n \subseteq \mathcal{Y}$, and $\mathcal{Y}|_{\preceq C\sigma}, D_1\theta_1, \dots, D_n\theta_n \models C\sigma$. Since $\mathcal{Y} \subseteq \mathcal{X}$, we deduce that $\forall i \in \{1 \dots n\}$, $\mathcal{Y}_i \subseteq \mathcal{X}$, hence $D_i \in S|_{\mathcal{X}}$. Thus, $\mathcal{Y}|_{\preceq C\sigma}, D_1\theta_1, \dots, D_n\theta_n \models C\sigma$, $\mathcal{Y}|_{\preceq C\sigma}, D_1\theta_1, \dots, D_n\theta_n \preceq C\sigma$ and $\mathcal{Y}|_{\preceq C\sigma} \cup \{D_1, \dots, D_n\} \subseteq S|_{\mathcal{X}} \cup \mathcal{X}$; we conclude that C is redundant w.r.t. $S|_{\mathcal{X}} \cup \mathcal{X}$. ∎

Theorem 4.19 *Let $\mathcal{X}$ be a satisfiable constraint in $\mathfrak{X}$ and let S be a set of standard clauses such that $S \models \mathcal{X}^{\mathrm{c}}$. If S^* is a c$\mathcal{SP}$-saturation of S, then there exists a constraint $\mathcal{Y} \subseteq \mathcal{X}$ such that $[\square \mid \mathcal{Y}] \in S^*$.*

Proof. Let $S' = S^*|_{\mathcal{X}} \cup \mathcal{X}$. We remark that S' is unsatisfiable: indeed, $S^*|_{\top} \models S$ since all the standard clauses in S must be redundant w.r.t. $S^*|_{\top}$; furthermore, $S^*|_{\top} \subseteq S^*|_{\mathcal{X}}$, so that $S' \models S^*|_{\top} \cup \mathcal{X} \models S \cup \mathcal{X} \models \{\mathcal{X}^{\mathrm{c}}\} \cup \mathcal{X}$.

We now prove that S' is saturated w.r.t. $\mathcal{SP}$ (assuming that the ordering $\prec$ is the same for both calculi and that if l is selected in $[C \mid \mathcal{X}]$ for c$\mathcal{SP}$, then l is selected in C for $\mathcal{SP}$). We only consider the case where the Superposition rule is applied, the proof for the other inference rules is similar. Let $C_1 = t \simeq r \vee P_1$ and $C_2 = u \bowtie v \vee P_2$ be two clauses occurring in S', where $t' = u|_p$, $\sigma = \mathrm{mgu}(t, t')$, $r\sigma \not\succeq t\sigma$, $v\sigma \not\succeq u\sigma$ and $(t \simeq r)\sigma$ and $(u \bowtie v)\sigma$ are selected in $C_1\sigma$ and $C_2\sigma$ respectively. Let $D \stackrel{\text{def}}{=} (u[r] \bowtie v \vee P_1 \vee P_2)\sigma$ be the clause generated by a Superposition inference from C_1 and C_2. We distinguish several cases.

- If both C_1 and C_2 occur in $S^*|_{\mathcal{X}}$, then S contains a c-clause of the form $[t \simeq r \vee P_1 \mid \mathcal{X}_1]$ and another of the form $[u \bowtie v \vee P_2 \mid \mathcal{X}_2]$, where $\mathcal{X}_1, \mathcal{X}_2 \subseteq \mathcal{X}$. It is clear that the c-Superposition rule applies on these c-clauses, yielding $[(u[r] \bowtie v \vee P_1 \vee P_2)\sigma \mid \mathcal{X}_1 \wedge \mathcal{X}_2]$. Since S^* is a c$\mathcal{SP}$-saturation of S by hypothesis, this c-clause is redundant w.r.t. S^*, and we deduce by Proposition 4.18 that D is redundant w.r.t. S'.
- If C_1 occurs in $S^*|_{\mathcal{X}}$ and C_2 occurs in $\mathcal{X}$, then $C_2 = u \bowtie v$ and S^* contains a c-clause of the form $[C_1 \mid \mathcal{X}_1]$ where $\mathcal{X}_1 \subseteq \mathcal{X}$. Then the Negative Assertion rule generates $[(u[r] \bowtie v \vee P_1) \mid \mathcal{X}_1 \wedge u \bowtie v]\sigma$, and since S^* is a c$\mathcal{SP}$-saturation of S, this clause is redundant w.r.t. S^*. Since $\mathcal{X}_1 \wedge u \bowtie v \subseteq \mathcal{X}$, by Proposition 4.18, we deduce that $(u[r] \bowtie v \vee P_1)\sigma$ is redundant w.r.t. S'.
- If C_1 occurs in $\mathcal{X}$ and C_2 occurs in $S^*|_{\mathcal{X}}$, then the proof is similar to the previous case, this time using the Positive Assertion rule.
- If both C_1 and C_2 occur in $\mathcal{X}$, then $\mathcal{X}$ cannot be a normalized constraint, since by Definition 3.1, $(u \bowtie v)^{\mathrm{c}}{}_{|\mathcal{X}^{\mathrm{c}} \setminus (u \bowtie v)^{\mathrm{c}}} \preceq (u[r] \bowtie v)^{\mathrm{c}} \prec (u \bowtie v)^{\mathrm{c}}$, which contradicts Point 1 of Definition 3.12.

Since S' is unsatisfiable and saturated, this set necessarily contains $\Box$ by refutational completeness of $\mathcal{SP}$, which entails that $\Box \in S^*|_{\mathcal{X}}$, hence the result. ■

The following theorem states the soundness and deductive completeness of c$\mathcal{SP}$, w.r.t. the set of implicates with a Skolemized negated form in $\mathfrak{X}$.

Theorem 4.20 *Let S be a set of standard clauses and S^* be a c$\mathcal{SP}$-saturation of S. For every $\mathcal{X} \in \mathfrak{X}$, $\mathcal{X}^{\mathrm{c}}$ is an implicate of S iff S^* contains a c-clause of the form $[\Box \,|\, \mathcal{X}']$ with $\mathcal{X}' \subseteq \mathcal{X}$.*

Proof. If $\mathcal{X}^{\mathrm{c}}$ is an implicate of S then $S \models \mathcal{X}^{\mathrm{c}}$ and Theorem 4.19 gives the required result. Conversely, if S^* contains a c-clause $[\Box \,|\, \mathcal{X}']$ where $\mathcal{X}' \subseteq \mathcal{X}$ then the result is obtained by using Lemma 4.15. ■

Remark 4.21 *Note that the fact that every constraint in $\mathfrak{X}$ is assumed to be normalized strongly restricts the search space. For instance, when $a \succ b \succ c \succ d \succ e$, no rule will apply on $[a \simeq b \,|\, c \simeq d]$ and $[a \not\simeq b \,|\, c \simeq e]$ because the obtained constraint $c \simeq d \wedge c \simeq e$ is not normalized.*

Example 4.22 *The following example shows how to derive the implicate $D = a \not\simeq d \vee f(c) \simeq f(b)$ from $\{c \simeq d, f(a) \simeq f(b)\}$, given the term ordering $d \prec c \prec b \prec a \prec f(d) \prec f(c) \prec f(b) \prec f(a)$.*

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

The negation of the constraint of the last c-clause $a \simeq d \wedge f(c) \not\simeq f(b)$ is the implicate D.

5. Practical Algorithms for Redundancy Detection

In Section 4.3, an abstract criterion was introduced to characterize c-clauses that are redundant in c$\mathcal{SP}$, i.e., that are useless for deriving implicates. While such a criterion is useful from a theoretical point of view, it does not provide any efficient way for storing the derived sets of c-clauses and for detecting redundant c-clauses. In the present section, we focus on ground clauses and we provide data-structures and algorithms for performing these tasks. It is important to notice that these algorithms may be used for different purposes, associated with slightly different (albeit strongly related) notions of redundancy.

- First, they can be used to store the generated implicates. Here, the stored objects are clauses (without constraints) and redundant clauses are exactly those clauses that are not prime or not normalized. A clause C is stored when the c-clause with empty clausal part $[\Box \,|\, \neg C]$ is derived.

- Second, they can be used to store constrained clauses generated during proof search. A c-clause is then considered redundant if it satisfies the conditions of Definition 5.2 below. In particular, both the clause part and constraint must be stored and the ordering restrictions must be taken into account.

In the remainder of the section, all the considered terms, clauses etc. are ground, unless specified otherwise.

Remark 5.1 *The well-known Herbrand theorem states that a set of first-order clauses S is unsatisfiable if and only if there exists a finite set S_g of ground instances of clauses in S that is unsatisfiable. There exist many algorithms to compute such a set S_g efficiently, effectively reducing first-order satisfiability to ground satisfiability*[4]*. These algorithms range from heuristic approaches (De Moura & Bjorner, 2007), used by SMT solvers with great practical success, to refutationally complete procedures (see, e.g., Ganzinger & Korovin, 2003; Ge & Moura, 2009), possibly for specific theories (Echenim & Peltier, 2012). Since S_g is a logical consequence of S, every implicate of S_g is also an implicate of S, and these procedures can be used also to generate implicates, at the cost of losing deductive-completeness (generating complete sets of implicates is infeasible in first-order logic anyway, as explained in Section 2).*

5.1 Constrained Clausal Trees

Instead of using the general notion of redundancy of Definition 4.9, which is hard to test efficiently in practice, we employ a one-to-one redundancy criterion which can be seen as a generalization of subsumption:

Definition 5.2 *A c-clause $[C\,|\,\mathcal{X}]$* C-subsumes *a c-clause $[D\,|\,\mathcal{Y}]$, written $[C\,|\,\mathcal{X}] \leq_{\mathrm{C}} [D\,|\,\mathcal{Y}]$, if $C \leq_{\mathrm{E}} D$, $C \preceq D$ and $\mathcal{X} \subseteq \mathcal{Y}$.*[5]

Proposition 5.3 *If $[C\,|\,\mathcal{X}] \leq_{\mathrm{C}} [D\,|\,\mathcal{Y}]$ then $[D\,|\,\mathcal{Y}]$ is redundant with respect to $\{[C\,|\,\mathcal{X}]\}$.*

Proof. Since $C \leq_{\mathrm{E}} D$, by Theorem 3.10 we have $C \models D$; hence the result. ∎

Note that, again, both parts of the c-clauses are handled in different ways: the inclusion relation $\subseteq$ used to compare constraints is clearly stronger than the E-subsumption relation $\leq_{\mathrm{E}}$ used for clauses, even enriched with an ordering constraint. For instance, assuming that $d \prec c \prec b \prec a$, we have $[d \not\simeq c \vee c \simeq a \mid \top] \leq_{\mathrm{C}} [d \not\simeq b \vee c \not\simeq b \vee b \simeq a \mid \top]$, but $[\square \mid d \simeq c \wedge c \not\simeq a] \not\leq_{\mathrm{C}} [\square \mid d \simeq b \wedge c \simeq b \wedge b \not\simeq a]$. As explained in Remark 4.11, comparing constraints using logical entailment would make the calculus incomplete.

Since all considered clauses are ground, they can systematically be replaced by the equivalent, normalized clauses; i.e., each c-clause $[C\,|\,\mathcal{X}]$ can be replaced by $[C_{\downarrow}\,|\,\mathcal{X}]$. This is justified by the fact that $C_{\downarrow}$ is equivalent to and smaller than C, thus $[C_{\downarrow}\,|\,\mathcal{X}]$ C-subsumes $[C\,|\,\mathcal{X}]$. We therefore assume that all the considered c-clauses are normalized and devise data-structures and algorithms for storing and retrieving them efficiently.

Definition 5.4 *We define a total ordering $<_\pi$ on literals as follows:*

4. Due to usual theoretical limitations, the generated set of clauses is infinite in general.
5. See Definition 4.9 for the abstract redundancy criterion and Definition 3.8 for the definition of $\leq_{\mathrm{E}}$.

- *positive literals are all greater than negative ones;*
- *if l_1 and l_2 are literals with the same polarity then $l_1 <_\pi l_2$ iff $l_1 \prec l_2$.*

Sets of constraints are stored in a trie data structure (Fredkin, 1960), formally defined as follows.

Definition 5.5 *A* constraint tree *is inductively defined as follows:*

- *$\top$ is a constraint tree.*
- *A given set of pairs $\{(l_i, T_i) \mid i \in \{1 \dots n\}\}$, such that $l_1, \dots, l_n$ are pairwise distinct ground literals and $T_1, \dots, T_n$ are constraint trees, is a constraint tree if, for every $i = 1, \dots, n$, each time T_i contains a pair (l', T'), we have $l_i <_\pi l'$.*

The set of constraints represented by a constraint tree T is denoted by $\mathcal{C}(T)$ and defined inductively as follows:

$$\mathcal{C}(T) \stackrel{def}{=} \begin{cases} \{\top\} & \text{if } T = \top, \\ \{l \wedge D \mid (l, T') \in T \wedge D \in \mathcal{C}(T')\} & \text{otherwise.} \end{cases}$$

Remark 5.6 *Employing the ordering $<_\pi$ to constrain the order in which literals occur along the branches of the trees has two uses:*

- *it limits the number of repetitions of the same literal,*
- *it simplifies the application of the redundancy elimination algorithms presented in Section 5.2.*

The first point can be enforced by using any total ordering on literals, but to ensure the second point, the use of $<_\pi$ is a necessity.

Note that by definition $\mathcal{C}(\emptyset) = \emptyset$. As implied by the definition, leaves of constraint trees can be either $\top$ or $\emptyset$, but in practice if a leaf is $\emptyset$ then the corresponding branch is irrelevant, because by definition two constraint trees that only differ by branches with leaves that are $\emptyset$ represent the same sets of constraints. In other words, a pair $(l, \emptyset)$ can be deleted from the tree T without affecting $\mathcal{C}(T)$. The only exception is the empty tree, for which the root is labeled with $\emptyset$.

The storage of constrained clauses is also based on a trie data-structure. It is similar to that of constraints, except that at each leaf of the tree storing the clauses, a constraint tree is appended to store the corresponding constraints.

Definition 5.7 *A* constrained clausal tree *or* c-tree *is inductively defined as follows:*

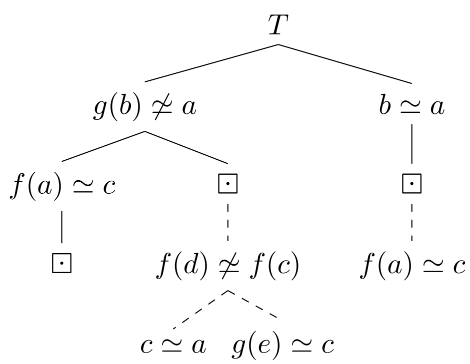
- *If T is a constraint tree then the set $\{(\boxdot, T)\}$ is a c-tree.*
- *If l is a literal and T is a c-tree then the set $\{(l, T)\}$ is a c-tree when for all $(l', T') \in T$, $l <_\pi l'$.*
- *If T_1, T_2 are c-trees such that every time $(l, T_1') \in T_1$ (resp. $(\boxdot, T_1') \in T_1$) and $(l, T_2') \in T_2$ (resp. $(\boxdot, T_2') \in T_2$) we have $T_1' = T_2'$, then $T_1 \cup T_2$ is a c-tree.*

The set of c-clauses represented by a c-tree T is denoted by $\mathcal{C}_c(T)$ and defined inductively as follows:

$$\mathcal{C}_c(T) = \begin{cases} \emptyset & \text{if } T = \emptyset \\ \{[\Box \,|\, \mathcal{X}] \mid \mathcal{X} \in \mathcal{C}(T')\} & \text{if } T = \{(\boxdot, T')\} \\ \{[l \vee C \,|\, \mathcal{X}] \mid [C \,|\, \mathcal{X}] \in \mathcal{C}_c(T')\} & \text{if } T = \{(l, T')\} \\ \mathcal{C}_c(T_1) \cup \mathcal{C}_c(T_2) & \text{if } T = T_1 \cup T_2,\ T_1 \neq \emptyset \text{ and } T_2 \neq \emptyset. \end{cases}$$

A c-tree T is normalized *if all the c-clauses in $\mathcal{C}(T)$ are normalized and non-tautological.*

Example 5.8 *The structure T in Figure 2 is a c-tree with the term order $a \prec b \prec c \prec g(c) \prec g(e) \prec f(c) \prec f(d)$. For readability, the labels are associated with the nodes rather than with the edges leading to them. Dotted lines denote the edges of the constraint trees occurring inside the c-tree.*



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

$[g(b) \not\simeq a \vee f(a) \simeq c \,|\, \top]$
$[g(b) \not\simeq a \,|\, f(d) \not\simeq f(c) \wedge c \simeq a]$
$[g(b) \not\simeq a \,|\, f(d) \not\simeq f(c) \wedge g(e) \simeq c]$
$[b \simeq a \,|\, f(a) \simeq c]$

Figure 2: A Constrained Clausal Tree

Definition 5.9 *Let T be a c-tree or a constraint tree.*

$$\text{size}(T) \stackrel{def}{=} \begin{cases} 0 & \text{if } T = \{(\boxdot, T')\} \text{ or } T = \{\top\}, \\ \displaystyle\sum_{(l,T') \in T} 1 + \text{size}(T') & \text{otherwise.} \end{cases}$$

Definition 5.10 *Let T be a c-tree and let C be a clause. Assume that $l <_\pi l'$ holds for all literals l and l' occurring in C and T respectively. Then $C.T$ is defined inductively as follows: $\Box.T \stackrel{def}{=} T$ and $(l \vee C).T \stackrel{def}{=} \{(l, (C.T))\}$ if $l = \min_{<_\pi}(l \vee C)$.*

5.2 Constrained Clausal Tree Operations

There are three main operations to perform on c-trees. The first one, called *forward* C-*subsumption*, consists in checking whether a new c-clause is C-subsumed by a c-clause already stored in a c-tree. The second one, called *backward* C-*subsumption*, removes from a c-tree all c-clauses that are C-subsumed by a given c-clause. The last one is the insertion of a new c-clause into a c-tree. This last operation is straightforward and thus will not be described here.

5.2.1 FORWARD C-SUBSUMPTION

The forward C-subsumption algorithm is called ISENTAILED (see Algorithm 2). The new c-clause is $[C \mid \mathcal{X}]$, the c-tree is T and the input clause N (initially empty) contains the literals occurring in the parent nodes in the recursive calls. It is also necessary to keep track of the negative literals of C in recursive calls after having used them a first time to rewrite literals in the c-tree. They are stored in the input clause M, which is also initially empty. During the traversal of the c-tree T, each branch (l, T') is dealt with differently depending on the relation between l and the considered c-clause C.

- If it is clear that $l \models M \vee C$, then the exploration of the branch continues on T'. This entailment condition is tested by checking that $l_{\downarrow M}$ is a contradiction (if l is negative) or that $C_{\downarrow M \vee l^c}$ contains a tautological literal (if l is positive). Such branches are respectively stored into the sets T_1 and T_3 at Lines 4 and 16.
- If the relation between l and $M \vee C$ is not currently determined (because l is $<_\pi$-greater than the literal currently considered in C and thus may entail literals that remain to be examined), then the minimal literal of C is added to M before restarting the exploration of the branch. Such branches are grouped in T_2 at Line 13.
- Lastly, if it is clear that $l \not\models C$, which is the case e.g. when l is $\leq_\pi$-smaller than all the literals in C, then the exploration of the branch is halted.

Note that the $\preceq$-test is reduced to a simple comparison between standard clauses (since the corresponding tree branch has already been explored), hence we do not detail the corresponding algorithm. In contrast, the $\subseteq$-test requires going through the constraint tree associated with the clausal branch that has just been explored. This test is performed by the ISINCLUDED routine, which is detailed in Algorithm 1. This algorithm is a lot simpler than ISENTAILED but its principle is the same, a depth-first traversal of the tree with recursive calls deleting the literals one after the other until the inclusion becomes obvious (or obviously false). In this algorithm, the branches $(l, T') \in T$ are grouped and dealt with in accordance with their relationship to $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$. If $l = m_1$ then only the inclusion of a branch of T' in $\mathcal{X} \setminus \{m_1\}$ is tested at Line 9. Otherwise if $l <_\pi m_1$ then clearly no branch of $l.T'$ is included in $\mathcal{X}$, and if $m_1 <_\pi l$, then it is possible that one of these branches is included in $\mathcal{X} \setminus \{m_1\}$, which is tested at Line 13.

Example 5.11 *To illustrate the functioning of these two algorithms, let us consider the c-tree T of Figure 2 and the two c-clauses $[C_1 \mid \mathcal{X}_1] = [f(b) \simeq f(a) \mid a \not\simeq d \wedge f(a) \simeq c]$ and $[C_2 \mid \mathcal{X}_2] = [g(b) \not\simeq a \mid c \simeq a]$. To test whether there are clauses in T that subsume $[C_1 \mid \mathcal{X}_1]$ and/or $[C_2 \mid \mathcal{X}_2]$, we call* ISENTAILED$([C_1 \mid \mathcal{X}_1], T, \square, \square)$ *and* ISENTAILED$([C_2 \mid \mathcal{X}_2], T, \square, \square)$ *which return **true** and **false** respectively. The inner workings of these two calls are described in Table 1 and 2 respectively (each line represents a recursive call).*

The correctness and termination of ISINCLUDED are stated in the following proposition.

Proposition 5.12 *Let $\mathcal{X}$ be a constraint and let T be a constraint tree. The call to* ISINCLUDED$(\mathcal{X}, T)$ *terminates and* ISINCLUDED$(\mathcal{X}, T) =$ ***true*** *iff there exists a constraint $\mathcal{Y} \in \mathcal{C}(T)$ such that $\mathcal{Y} \subseteq \mathcal{X}$.*

Call	Assignments & tests	Next call(s)	
$\textsc{isEntailed}([C_1 \,\vert\, \mathcal{X}_1], T, \Box, \Box)$	$T_1 \leftarrow \emptyset$ $m_1 \leftarrow f(b) \simeq f(a)$ $T_3 \leftarrow \{(b \simeq a, T_\beta)\}$	line 17	(1)
$\rightarrow$ $\textsc{isEntailed}([C_1 \,\vert\, \mathcal{X}_1], T_\beta, \Box, b \simeq a)$	$(\boxdot, T'_\beta) \in T_\beta$ $b \simeq a \preceq f(b) \simeq f(a)$	Line 1	
$\rightarrow\rightarrow$ $\textsc{isIncluded}(\mathcal{X}_1, T'_\beta)$	$m_1 \leftarrow a \not\simeq d$ $T_1 \leftarrow \emptyset$ $T_2 \leftarrow T'_\beta$	Line 13	(2)
$\rightarrow\rightarrow\rightarrow$ $\textsc{isIncluded}(f(a) \simeq c, T'_\beta)$	$m_1 \leftarrow f(a) \simeq c$ $T_1 \leftarrow T'_\beta$	Line 9	
$\rightarrow\rightarrow\rightarrow\rightarrow$ $\textsc{isIncluded}(\top, \top)$		returns **true** (Line 2)	(3)

Comments	
(1)	The left child of T is not used in recursive calls since $g(b) \not\simeq a$ does not entail C_1 (line 16). T_β is the subtree of T below $b \simeq a$.
(2)	Since T_1 is empty, there is no recursive call line 9. T'_β is the only child of T_β.
(3)	This value is propagated up to the initial call, that also returns **true**.

Table 1: Running of the Call $\textsc{isEntailed}([C_1 \,\vert\, \mathcal{X}_1], T, \Box, \Box)$

Algorithm 1 $\textsc{isIncluded}(\mathcal{X},\ T)$

Require: T is a constraint tree, $\mathcal{X}$ is a constraint.
Ensure: $\textsc{isIncluded}(\mathcal{X},\ T) =$ **true** iff $\exists \mathcal{Y} \in \mathcal{C}(T)$, $\mathcal{Y} \subseteq \mathcal{X}$.

1: **if** $T = \top$ **then**
2:     **return true**
3: **end if**
4: **if** $\mathcal{X} = \top$ **then**
5:     **return false**
6: **end if**
7: $m_1 \leftarrow \min_{<_\pi} \{m \in \mathcal{X}\}$
8: $T_1 \leftarrow \{(l, T') \in T \mid l = m_1\}$
9: **if** $\bigvee_{(l,T') \in T_1} \textsc{isIncluded}(\mathcal{X} \setminus \{m_1\}\,,\ T')$ **then**
10:     **return true**
11: **end if**
12: $T_2 \leftarrow \{(l, T') \in T \mid m_1 <_\pi l\}$
13: **return** $\bigvee_{(l,T') \in T_2} \textsc{isIncluded}(\mathcal{X} \setminus \{m_1\}\,,\ l.T')$

Proof. The termination of $\textsc{isIncluded}(\mathcal{X}, T)$ is due to the fact that $|\mathcal{X}|$ decreases at each recursive call. To prove the equivalence property, we show each implication by induction on $|\mathcal{X}|$.

Call	Assignments & tests	Next call(s)	
isEntailed$([C_2 \,\vert\, \mathcal{X}_2], T, \Box, \Box)$	$T_1 \leftarrow \emptyset$ $m_1 \leftarrow g(b) \not\simeq a$ $T_2 \leftarrow T$	Line 14 ($\times 2$)	(1)
$\rightarrow$ isEntailed$([\Box \,\vert\, \mathcal{X}_2], \{(g(b) \not\simeq a, T_\alpha)\}, g(b) \not\simeq a, \emptyset)$	$T_1 \leftarrow \{(g(b) \not\simeq a, T_\alpha)\}$	Line 5	(2)
$\rightarrow\rightarrow$ isEntailed$([\Box \,\vert\, \mathcal{X}_2], T_\alpha, g(b) \not\simeq a, g(b) \not\simeq a)$	$(\boxdot, T'_\alpha) \in T_\alpha$ $g(b) \not\simeq a \preceq g(b) \not\simeq a$	Line 1	(3)
$\rightarrow\rightarrow\rightarrow$ isIncluded$(\mathcal{X}_2, T'_\alpha)$	$m_1 \leftarrow c \simeq a$ $T_1 \leftarrow \emptyset$ $T_2 \leftarrow \emptyset$	returns **false** (Line 13)	(4)
$\rightarrow\rightarrow$ (continued)	$T_1 \leftarrow \emptyset$	returns **false** (Line 9)	(5)
$\rightarrow$ (continued)		returns **false** (Line 9)	(6)
$\rightarrow$ isEntailed$([\Box \,\vert\, \mathcal{X}_2], \{(b \simeq a, T_\beta)\}, g(b) \not\simeq a, \emptyset)$	$T_1 \leftarrow \emptyset$	returns **false** (Line 9)	(7)

Comments	
(1)	Since T has two children, there are two recursive calls performed line 14.
(2)	T_α is the subtree of T below $g(b) \not\simeq a$.
(3)	T'_α is the child of T_α below $\boxdot$.
(4)	This value is propagated up to the previous call.
(5)	Since the call Line 1 returned false, Line 8 is triggered next. This value is propagated up to the previous call.
(6)	This value is propagated up to the main call.
(7)	T_β is the subtree of T below $b \simeq a$. The result is propagated up to the main call.

Table 2: Running of the Call isEntailed$([C_2 \,|\, \mathcal{X}_2], T, \Box, \Box)$

Assume that isIncluded$(\mathcal{X}, T) =$ **true**. If $T = \top$ then $\mathcal{C}(T) = \{\top\}$ and $\top \subseteq \mathcal{X}$. Otherwise $\mathcal{X}$ cannot be empty or the instruction at Line 5 would be triggered. Let $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$. If the call terminates at Line 10 then there exists a pair $(l, T') \in T_1$ such that isIncluded$(\mathcal{X} \setminus \{m_1\}, T')$ returns **true**. By definition of T_1, we must have $l = m_1$, and by the induction hypothesis there exists $\mathcal{Y} \in \mathcal{C}(T')$ such that $\mathcal{Y} \subseteq \mathcal{X} \setminus \{m_1\}$, hence $\mathcal{Y} \wedge m_1 \subseteq \mathcal{X}$. If the call terminates at Line 13, then there exists a pair $(l, T') \in T_2 \subseteq T$ such that isIncluded$(\mathcal{X} \setminus \{m_1\}, l.T')$ returns **true**. By the induction hypothesis, there exists a constraint $\mathcal{Y} \in \mathcal{C}(l.T')$ such that $\mathcal{Y} \subseteq \mathcal{X} \setminus \{m_1\}$, hence $\mathcal{Y} \in \mathcal{C}(T)$ and $\mathcal{Y} \subseteq \mathcal{X}$.

For the converse implication, let us assume that T and $\mathcal{X}$ are such that there exists a constraint $\mathcal{Y} \in \mathcal{C}(T)$, where $\mathcal{Y} \subseteq \mathcal{X}$. If $\mathcal{X} = \top$ then necessarily $\mathcal{Y} = \top$. By definition of a constraint tree we must have $T = \top$, and **true** is returned at Line 2. Otherwise $\mathcal{X}$ is not empty and we let $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$. If $T = \top$, then the return statement at Line 2 is triggered again and we have the result. Otherwise, we define $l_1 = \min_{<_\pi} \{l \in \mathcal{Y}\}$. Given the ordering constraints imposed on constraint trees, there necessarily exists a pair $(l_1, T') \in T$ such that $\mathcal{Y} \setminus \{l_1\} \in \mathcal{C}(T')$. We distinguish several cases.

- If $l_1 = m_1$ then $(l_1, T') \in T_1$ and $\mathcal{Y} \setminus \{l_1\} \subseteq \mathcal{X} \setminus \{m_1\}$. Hence by the induction hypothesis $\textsc{isIncluded}(\mathcal{X} \setminus \{m_1\}, T')$ returns **true**, the test at Line 9 succeeds and $\textsc{isIncluded}(\mathcal{X}, T)$ also returns **true**.
- If $l_1 <_\pi m_1$ then we cannot have $\mathcal{Y} \subseteq \mathcal{X}$, and this contradicts the initial hypothesis.
- If $m_1 <_\pi l_1$ then $(l_1, T') \in T_2$ and we must have $\mathcal{Y} \subseteq \mathcal{X} \setminus \{m_1\}$. By the induction hypothesis $\textsc{isIncluded}(\mathcal{X} \setminus \{m_1\}, l_1.T')$ returns **true**, and therefore, $\textsc{isIncluded}(\mathcal{X}, T)$ returns **true** at Line 13. ■

Algorithm 2 $\textsc{isEntailed}([C \,|\, \mathcal{X}],\ T,\ M,\ N)$

Require: T is a c-tree, M is negative, $M \vee C$ is a normalized non-tautological clause and $N \models M \vee C$

Ensure: $\textsc{isEntailed}([C \,|\, \mathcal{X}],\ T,\ M,\ N) =$ **true** iff there exists $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T)$ such that $[D \vee N \,|\, \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \,|\, \mathcal{X}]$

if $(\boxdot, T') \in T \wedge (N \preceq (M \vee C)) \wedge \textsc{isIncluded}(\mathcal{X}, T')$ **then**
    return true
end if
$T_1 \leftarrow \{(l, T') \in T \mid l_{\downharpoonright M}$ is a contradiction$\}$
if $\bigvee_{(l,T') \in T_1} \textsc{isEntailed}([C \,|\, \mathcal{X}],\ T',\ M,\ N \vee l)$ **then**
    return true
end if
if $C = \square$ **then**
    return false
end if
$m_1 \leftarrow \min_{<_\pi} \{m \in C\}$
if m_1 is of the form $u \not\simeq v$, with $u \succ v$ **then**
    $T_2 \leftarrow \{(l, T') \in T \mid l_{\downharpoonright M} \not<_\pi m_1$ and $\nexists w, (l_{\downharpoonright M} = u \not\simeq w$, with $u \succ w \succ v)\}$
    return $\bigvee_{(l,T') \in T_2} \textsc{isEntailed}([C \setminus \{m_1\} \,|\, \mathcal{X}],\ l.T',\ M \vee m_1,\ N)$
else
    $T_3 \leftarrow \{(l, T') \in T \mid C_{\downharpoonright M \vee l^c}$ contains a tautological literal$\}$
    return $\bigvee_{(l,T') \in T_3} \textsc{isEntailed}([C \,|\, \mathcal{X}],\ T',\ M,\ N \vee l)$
end if

The correctness of $\textsc{isEntailed}$ is stated in Theorem 5.16. The next propositions are steps of this proof. Proposition 5.14 shows that all the candidate branches starting with a negative literal are necessarily in $T_1 \cup T_2$, and for T_2, Proposition 5.15 justifies the restriction imposed on the form of the first literal of the selected branches. Both propositions are based on the following result:

Proposition 5.13 *If C and D are clauses such that $C \vee D$ is normalized, then C is also normalized.*

Proof. This is due to the fact that since $\equiv_C \subseteq \equiv_{C \vee D}$, if $l \in C$ then $l = l_{\downarrow C \vee D \setminus l} \preceq l_{\downarrow C \setminus l} = l$, and if l and m are such that $m_{\downarrow l^c \vee C^-}$ is a tautology then necessarily $m_{\downarrow l^c \vee C^- \vee D^-}$ is also a tautology, which is impossible by definition. ∎

Proposition 5.14 *Let C be a non-empty clause such that $M \vee C$ is normalized. Let l be a literal such that $l_{\downarrow M} \models M \vee C$, and let $m_1 = \min\limits_{<_\pi} \{m \in C\}$. If l is a negative literal, then either $l_{\downarrow M}$ is a contradiction or $l_{\downarrow M} \not<_\pi m_1$.*

Proof. Assume that $l_{\downarrow M}$ is of the form $u \not\simeq v$ and that $l_{\downarrow M} <_\pi m_1$. Since $l_{\downarrow M} \models M \vee C$, by Theorem 3.10, $l_{\downarrow M} \leq_{\mathrm{E}} M \vee C$, so that $u \equiv_{M \vee C} v$ and $u_{\downarrow M \vee C} = v_{\downarrow M \vee C}$.

- If m_1 is of the form $u \not\simeq v'$ where $v' \prec u$, then $u_{\downarrow M \vee C} = v'$ because $M \vee C$ is normalized, hence we also have $v_{\downarrow M \vee C} = v'$, so that $v' \preceq v$. We deduce that $m_1 \preceq l_{\downarrow M}$, which contradicts the assumption that $l_{\downarrow M} <_\pi m_1$.
- Otherwise, m_1 is either a positive literal, or of the form $s \not\simeq t$, where $s \succ u$ and $s \succeq t$. Thus in both cases $u_{\downarrow C} = u$ and $v_{\downarrow C} = v$ (because the smallest term to be potentially rewritten by the ground rewriting system associated[6] with C, namely s, is greater than u, see Proposition 3.17), thus $u = u_{\downarrow M} = u_{\downarrow M \vee C} = v_{\downarrow M \vee C} = v_{\downarrow M} = v$. ∎

Proposition 5.15 *Let $M \vee C$ be a normalized clause where M is negative, and assume that $m_1 = \min\limits_{<_\pi} \{m \in C\}$ is a negative literal $u \not\simeq v$ with $u \succ v$. Let l be a literal such that $l_{\downarrow M} \not<_\pi m_1$ and $l_{\downarrow M} \models M \vee C$. The literal $l_{\downarrow M}$ cannot be of the form $u \not\simeq w$ with $u \succ w \succ v$*

Proof. Assume that $l_{\downarrow M}$ is of the form $u \not\simeq w$ with $u \succ w \succ v$. Then because $M \vee C$ is normalized, $u_{\downarrow M \vee C} = v$. In addition, by Theorem 3.10, since $l_{\downarrow M} \models M \vee C$, we have $w_{\downarrow C \vee M} = u_{\downarrow C \vee M}$, thus $w_{\downarrow M \vee C} = v$. Now by Proposition 3.17, since $w \prec u$ and all the maximal terms in the literals of C^- are strictly greater than w, the latter cannot be rewritten by the rewriting system associated with C. We deduce that $w_{\downarrow C \vee M} = w_{\downarrow M} = w$; hence $w = v$, which contradicts the hypothesis that $w \succ v$. ∎

Theorem 5.16 *Given a c-clause $[C \mid \mathcal{X}]$ and a c-tree T, let M and N be clauses such that M is negative, $M \vee C$ is normalized and $N \models M \vee C$. The call* isEntailed$([C \mid \mathcal{X}], T, M, N)$ *terminates. Furthermore,* isEntailed$([C \mid \mathcal{X}], T, M, N) =$ ***true*** *iff there exists $[D \mid \mathcal{Y}] \in \mathcal{C}_c(T)$ such that $[D \vee N \mid \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \mid \mathcal{X}]$.*

Proof. The termination proof is trivial, because at all recursive calls, the positive value $|C| + \mathrm{size}(T)$ strictly decreases.

The correction proof requires two inductions, one for each implication. For the left-to-right direction, the proof consists in going through the different cases enumerated by the algorithm to verify that the requirements of the recursive calls are indeed met and that it is possible to derive the required property from its inductive children. In the other direction the different cases that validate the right-hand side of the equivalence are considered and matched with the different cases of the algorithm.

6. See Definition 3.16.

Left-to-Right Direction. If isEntailed($[C \,|\, \mathcal{X}], T, M, N$) is **true**, then one of the **return** statements is triggered and returns **true**. We consider each of them in their order of appearance.

1. Line 2, $(\boxdot, T') \in T$, $N \preceq M \vee C$ and isIncluded($\mathcal{X}, T'$) returns **true**. Since isIncluded is correct by Proposition 5.12, we deduce that $\exists \mathcal{Y} \in \mathcal{C}(T')$ such that $\mathcal{Y} \subseteq \mathcal{X}$. In this case $[\square \mid \mathcal{Y}] \in \mathcal{C}_c(T)$, and since $N \models M \vee C$ by hypothesis, the property $[N \,|\, \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \,|\, \mathcal{X}]$ is verified.

2. Line 5, T_1 is set to $\{(l, T') \in T \mid l_{\downharpoonleft M} \text{ is a contradiction}\}$ and **true** is returned by the disjunction $\bigvee_{(l,T') \in T_1}$ isEntailed($[C \mid \mathcal{X}], T', M, N \vee l$). Thus there exists a pair $(l, T') \in T_1$ such that isEntailed($[C \mid \mathcal{X}], T', M, N \vee l$) returns **true** and $l_{\downharpoonleft M}$ is a contradiction. By Theorem 3.10, $l \models M$, thus $l \models M \vee C$. Moreover by hypothesis $N \models M \vee C$, hence these two properties ensure that the preconditions of the corresponding recursive call are met. By induction there exists $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T')$ such that $[D \vee l \vee N \,|\, \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \,|\, \mathcal{X}]$. Since $[l \vee D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T)$, the result holds.

3. Line 14, $C \neq \square$ and $\bigvee_{(l,T') \in T_2}$ isEntailed($[C \setminus m_1 \mid \mathcal{X}], l.T', M \vee m_1, N$) returns **true**, where $m_1 = \min_{<_\pi} \{m \in C\}$ is of the form $u \not\simeq v$ with $u \succ v$ and T_2 is the set $\{(l, T') \in T \mid l_{\downharpoonleft M} \not<_\pi m_1 \text{ and } \nexists w, (l_{\downharpoonleft M} = u \not\simeq w, \text{ with } u \succ w)\}$. Thus there is a pair $(l, T') \in T$ such that $l_{\downharpoonleft M} \not<_\pi m_1$ and $l_{\downharpoonleft M}$ is not of the form $u \not\simeq w$ with $u \succ w$, for which isEntailed($[C \setminus m_1 \,|\, \mathcal{X}], l.T', M \vee m_1, N$) returns **true**. By hypothesis $N \models M \vee C$; furthermore, $M \vee C = M \vee m_1 \vee (C \setminus m_1)$ and $M \vee m_1$ is negative. Therefore the preconditions of this recursive call are satisfied. By induction, its returning **true** entails that there exists $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(l.T')$ such that $D \vee N \leq_{\mathrm{C}} M \vee m_1 \vee C \setminus m_1$. Since $\mathcal{C}_c(l.T') \subseteq \mathcal{C}_c(T)$, the c-clause $[D \,|\, \mathcal{Y}]$ also belongs to $\mathcal{C}_c(T)$, whence the result.

4. Line 17, $m_1 = \min_{<_\pi} \{m \in C\}$ is of the form $u \simeq v$, l is positive and the disjunction $\bigvee_{(l,T') \in T_3}$ isEntailed($[C \,|\, \mathcal{X}], T', M, N \vee l$) returns true, with T_3 equal to the set $\{(l, T') \in T \mid C_{\downharpoonleft M \vee l^c} \text{ contains a tautological literal}\}$. In this case there exist $(l, T') \in T$ and $m_2 \in C$ such that $m_{2\downharpoonleft M \vee l^c}$ is a tautology, thus, by Theorem 3.10, $l \models M \vee m_2$ and $l \models M \vee C$. Since $N \models M \vee C$, the preconditions of the corresponding recursive call are met. Thus by the induction hypothesis $[D \vee l \vee N \,|\, \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \,|\, \mathcal{X}]$ and $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T')$. Since $[l \vee D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T)$, we have the result.

Right-to-Left Direction. Assume that there exists a c-clause $[D \,|\, \mathcal{Y}]$ in $\mathcal{C}_c(T)$ such that $[D \vee N \mid \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \mid \mathcal{X}]$, where C, T, M and N respect the preconditions of the algorithm. We consider several cases.

- If $D = \square$ then $(\boxdot, T') \in T$ and Line 2 is reached. Since $[D \vee N \,|\, \mathcal{Y}] \leq_{\mathrm{C}} [M \vee C \,|\, \mathcal{X}]$ we deduce that $\mathcal{Y} \subseteq \mathcal{X}$ and $N \preceq M \vee C$. Consequently, the tests $N \preceq M \vee C$ is true and isIncluded($\mathcal{X}, T'$) is true by Proposition 5.12, thus isEntailed($[C \mid \mathcal{X}], T, M, N$) returns **true**.

- Otherwise D is of the form $l \vee D'$, with $(l, T') \in T$ and $[D' \,|\, \mathcal{Y}] \in \mathcal{C}_c(T')$.

– If $C = \Box$, then since $[l \vee D' \vee N \,|\, \mathcal{Y}] \leq_{\rm C} [M \,|\, \mathcal{X}]$, in particular $l \vee D' \vee N \models M$. Theorem 3.10 guarantees that $\equiv_D \subseteq \equiv_M$, and since M is negative, D cannot contain any positive literal by Condition 2 of Definition 3.8. This means that $l_{\downharpoonright M}$ must be a contradiction, and thus $(l, T') \in T_1$. It is straightforward to verify that the preconditions of isEntailed$(C, T', M, N \vee l)$ at Line 5 are met, thus by the induction hypothesis, this call returns **true**, and isEntailed$([C \,|\, \mathcal{X}], T, M, N)$ also returns **true** at Line 6.

– If C is of the form $m_1 \vee C'$ with $m_1 = \min_{<_\pi} \{m \in C\}$ and l is a negative literal, then since $l \models M \vee C$ and $l \equiv_{M \vee C} l_{\downharpoonright M}$ by Proposition 3.5, we deduce that $l_{\downharpoonright M} \models M \vee C$. By Proposition 5.14, either 1) $l_{\downharpoonright M}$ is a contradiction, or 2) $l_{\downharpoonright M} \not<_\pi m_1$.

1. If $l_{\downharpoonright M}$ is a contradiction then $(l, T') \in T_1$ and since $l \vee D' \vee N \models M \vee C$ by hypothesis, we have $N \vee l \models M \vee C$; thus the recursive call isEntailed$([C \,|\, \mathcal{X}], T', M, N \vee l)$ at Line 6 returns **true** by the induction hypothesis.
2. If $l_{\downharpoonright M} \not<_\pi m_1$ then m_1 must be a negative literal by definition of $<_\pi$, and since $N \vee l \vee D' \models M \vee C$, we have $N \vee l \vee D' \models M \vee m_1 \vee (C \setminus m_1)$. By Proposition 5.15, $(l, T') \in T_2$. Since $M \vee C$ is normalized, so is $M \vee m_1 \vee (C \setminus m_1)$, thus the preconditions of isEntailed at Line 14 are met and the call returns **true** by the induction hypothesis.

– Now assume that $C = m_1 \vee C'$, where $m_1 = \min_{<_\pi} \{m \in C\}$, and that l is positive. If m_1 is negative then $(l, T') \in T_2$, Line 14 is reached and returns **true** as in the previous case. Otherwise by Theorem 3.10 there exists a positive literal m_2 in C such that $m_{2\downharpoonright M \vee C \vee l^c}$ is a tautology. But since m_1 is positive, by definition of $<_\pi$, C must be a positive clause and $m_{2\downharpoonright M \vee C \vee l^c} = m_{2\downharpoonright M \vee l^c}$. This implies that $(l, T') \in T_3$. Since $[N \vee l \vee D' \,|\, \mathcal{Y}] \leq_{\rm C} [M \vee C \,|\, \mathcal{X}]$, the preconditions for the call to isEntailed$([C \,|\, \mathcal{X}], T', M, N \vee l)$ at Line 17 are met and by the induction hypothesis, this call returns **true**. ∎

5.2.2 Backward c-Subsumption.

Backward c-Subsumption handles the removal from a c-tree T of all the c-clauses $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T)$ that are c-subsumed by a given c-clause $[C \,|\, \mathcal{X}]$, under the assumption that $[C \,|\, \mathcal{X}]$ itself is not c-subsumed by a c-clause stored in T. The principle of pruneEntailed (Algorithm 5) is very similar to that of the isEntailed algorithm: it is a depth-first traversal of T with a rewriting of literals. The main difference is that the roles of $[C \,|\, \mathcal{X}]$ and T are reversed: the literals of a branch of T are rewritten using the negative literals of C. When the c-subsumption test succeeds, the algorithm cuts the corresponding branch in T before exploring the remaining branches. In this algorithm, the clause N stores the already explored negative literals in the branch of T under investigation and several cases are distinguished depending on the form of the minimum literal m_1 in C. The branches ending with $\emptyset$ are systematically deleted from the output tree. The role of algorithm pruneInf (Algorithm 4) is to complete the exploration of each clausal branch of the c-tree so that the $\preceq$-comparison between these and C can be done (as explained in the previous section, the incompatibility of $<_\pi$ and $\preceq$ prevents the $\preceq$-test from being performed directly on the tree). In practice, once the test $C \not\preceq N$ fails, it also fails in all the subsequent recursive calls. Thus a sub-procedure containing only Lines 1 and 5 of pruneInf is used from this point on, producing the same result more efficiently. Finally, algorithm pruneIncluded (Algorithm 3) can be seen as a simpler version of pruneEntailed, where inclusion is tested instead of e-subsumption. It is the counterpart of algorithm isIncluded used in isEntailed, and is based on the same principle but with a swap in the roles of the constraint and the constraint tree.

Example 5.17 *To illustrate how these three algorithms work, let us consider again the c-tree T of Figure 2 and the c-clause $[C_2 \mid \mathcal{X}_2] = [g(b) \not\simeq a \mid c \simeq a]$, which is such that* isEntailed$([C_2 \mid \mathcal{X}_2], T, \square, \square) = \bot$ *as seen in Example 5.11. To remove the clauses in T that are subsumed by $[C_2 \mid \mathcal{X}_2]$, we call* pruneEntailed$([C_2 \mid \mathcal{X}_2], T, \square, \square)$. *The result is the c-tree T without the branch for the c-clause $[g(b) \not\simeq a \mid f(d) \not\simeq f(c) \vee c \simeq a]$, which is the only c-clause in T that is redundant w.r.t. $[C_2 \mid \mathcal{X}_2]$. The inner working of this call is described in Table 3.*

The correction of pruneEntailed is stated in Theorem 5.20. The intermediate termination and correction results concerning pruneIncluded and pruneInf are stated respectively in Proposition 5.18 and Proposition 5.19.

Algorithm 3 pruneIncluded$(\mathcal{X},\ T)$

Require: T is a constraint tree, $\mathcal{X}$ is a constraint.
Ensure: $\mathcal{C}(T_{out}) = \{\mathcal{Y} \in \mathcal{C}(T) | \mathcal{X} \not\subseteq \mathcal{Y}\}$, where $T_{out} =$ pruneIncluded$(\mathcal{X},\ T)$.

if $\mathcal{X} = \top$ **then**
    return $\emptyset$
end if
if $T = \top$ **then**
    return T
end if
$m_1 \leftarrow \min_{<_\pi} \{m \in \mathcal{X}\}$
$T_1 \leftarrow \{(l, T') \in T \mid l = m_1\}$
$T_{out1} \leftarrow \{(l, \text{pruneIncluded}(\mathcal{X} \setminus \{m_1\}\ ,\ T')) \mid (l, T') \in T_1$
        $\wedge\ \text{pruneIncluded}(\mathcal{X} \setminus \{m_1\}\ ,\ T') \neq \emptyset\}$
$T_2 \leftarrow \{(l, T') \in T \setminus T_1 \mid l <_\pi m_1\}$
$T_{out2} \leftarrow \{(l, \text{pruneIncluded}(\mathcal{X},\ T')) \mid (l, T') \in T_2$
        $\wedge\ \text{pruneIncluded}(\mathcal{X},\ T') \neq \emptyset\}$
return $T_{out1} \cup T_{out2} \cup (T \setminus (T_1 \cup T_2))$

Proposition 5.18 *Let T be a constraint tree and let $\mathcal{X}$ be a constraint. The algorithm call* pruneIncluded$(\mathcal{X}, T)$ *terminates and if $T_{out} =$* pruneIncluded$(\mathcal{X}, T)$, *then $\mathcal{C}(T_{out}) = \{\mathcal{Y} \in \mathcal{C}(T) \mid \mathcal{X} \not\subseteq \mathcal{Y}\}$.*

Proof. The termination of pruneIncluded is the consequence of the strict decrease of size(T) at each recursive call. Let $T_{out} =$ pruneIncluded$(\mathcal{X}, T)$. If $T_{out} = \emptyset$ then $\mathcal{C}(T_{out}) =$

Call	Assignments and tests	Next call(s)	
$\textsc{pruneEntailed}([C_2 \,\vert\, \mathcal{X}_2], T, \square, \square)$	$m_1 \leftarrow g(b) \not\simeq a$ $T_{\boxdot} \leftarrow \emptyset$ $T_1 \leftarrow \{(g(b) \simeq a, T_\alpha)\}$	Line 10	(1)
$\rightarrow$ $\textsc{pruneEntailed}([C_2 \,\vert\, \mathcal{X}_2], T_\alpha, \square,$ $g(b) \not\simeq a)$	$m_1 \leftarrow g(b) \not\simeq a$	Line 6	
$\rightarrow\rightarrow$ $\textsc{pruneEntailed}([\square \,\vert\, \mathcal{X}_2], T_\alpha,$ $g(b) \not\simeq a, g(b) \not\simeq a)$		Line 2	
$\rightarrow\rightarrow\rightarrow$ $\textsc{pruneInf}([C_2 \,\vert\, \mathcal{X}_2], T_\alpha, g(b) \not\simeq a)$		Line 1	
$\rightarrow\rightarrow\rightarrow\rightarrow$ $\textsc{pruneInf}([C_2 \,\vert\, \mathcal{X}_2], T_\gamma,$ $g(b) \not\simeq a \vee f(a) \simeq c)$	$T_{out1} \leftarrow \emptyset$ $C_2 \preceq g(b) \not\simeq a \vee f(a) \simeq c$	Line 5	(2)
$\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow$ $\textsc{pruneIncluded}(\mathcal{X}_2, \top)$		none	(3)
$\rightarrow\rightarrow\rightarrow$ (continued)	$T_{out1} \leftarrow T_\gamma$ $C_2 \preceq g(b) \not\simeq a$	Line 5	
$\rightarrow\rightarrow\rightarrow\rightarrow$ $\textsc{pruneIncluded}(\mathcal{X}_2, T'_\alpha)$	$m_1 \leftarrow c \simeq a$ $T_1 \leftarrow \emptyset$ $T_2 \leftarrow \{(f(d) \not\simeq f(c), T'_\delta)\}$	Line 11	(4)
$\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow$ $\textsc{pruneIncluded}(\mathcal{X}_2, T'_\delta)$	$m_1 \leftarrow c \simeq a$ $T_1 \leftarrow \{(c \simeq a, \top)\}$	Line 9	
$\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow$ $\textsc{pruneIncluded}(\top, \top)$			(5)
$\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow$ (continued)	$T_2 \leftarrow \emptyset$		(6)

Comments	
(1)	T_α is the subtree of T below the literal $g(b) \not\simeq a$. The other child of T is not modified.
(2)	$T_\gamma = (\boxdot, \top)$ is the subtree of T_α below the rightmost literal $f(a) \simeq c$.
(3)	The tree is not pruned since $\mathcal{X}_2 \neq \top$.
(4)	$(\boxdot, T'_\alpha) \in T_\alpha$, T'_δ is the child of T'_α.
(5)	This branch is pruned: $\emptyset$ is returned and discarded in the previous call.
(6)	The branch containing $g(e) \simeq c$ is left unchanged since it belongs to $T'_\delta \setminus (T_1 \cup T_2)$.

Table 3: Running of the Call $\textsc{pruneEntailed}([C_2 \,\vert\, \mathcal{X}_2], T, \square, \square)$

$\emptyset$ and the proposition holds no matter the original values of T and $\mathcal{X}$. Otherwise, if $T_{out} = \{\top\}$ then $\mathcal{C}(T_{out}) = \{\top\}$ and the proposition is true only if $\mathcal{X} \neq \top$. Since Line 2 is triggered when $\mathcal{X} = \top$ before anything else, necessarily $T_{out} = \emptyset$, a contradiction with the current hypothesis.

In the general case, we consider the two inclusions separately.

Left-in-Right Inclusion. Let us consider the case $\mathcal{C}(T_{out}) \subseteq \{\mathcal{Y} \in \mathcal{C}(T) \mid \mathcal{X} \not\subseteq \mathcal{Y}\}$, by taking $\mathcal{Y} \in \mathcal{C}(T_{out})$ such that $\mathcal{Y} = l \vee \mathcal{Y}'$, where $(l, T'_{out}) \in T_{out}$ and $\mathcal{Y}' \in \mathcal{C}(T'_{out})$. This is sufficient since the base cases are covered in the previous paragraph. Since $T_{out} \neq \emptyset$, we know that $\mathcal{X} \neq \top$ thus $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$ exists. There are three cases to examine:

- If $\mathcal{Y} \in \mathcal{C}(T_{out1})$ then $T'_{out} = \textsc{pruneIncluded}(\mathcal{X} \setminus \{m_1\}, T')$ where $(l, T') \in T_1$, hence $l = m_1$. By the induction hypothesis $\mathcal{Y}' \in \mathcal{C}(T')$ and $\mathcal{X} \setminus \{m_1\} \nsubseteq \mathcal{Y}'$, thus $\mathcal{X} \nsubseteq m_1 \wedge \mathcal{Y}'(= \mathcal{Y})$ and $\mathcal{Y} \in \mathcal{C}(T_1) \subseteq \mathcal{C}(T)$.

- If $\mathcal{Y} \in \mathcal{C}(T_{out2})$ then $T'_{out} = \textsc{pruneIncluded}(\mathcal{X}, T')$ where $(l, T') \in T_2$ thus $l <_\pi m_1$. By the induction hypothesis $\mathcal{Y}' \in \mathcal{C}(T')$ thus $\mathcal{Y} \in \mathcal{C}(T)$ and $\mathcal{X} \nsubseteq \mathcal{Y}'$, thus in particular $m_1 \notin \mathcal{Y}'$ hence $m_1 \notin l \vee \mathcal{Y}'$ ensuring that $\mathcal{X} \nsubseteq \mathcal{Y}$.

- If $\mathcal{Y} \in \mathcal{C}(T \setminus (T_1 \cup T_2))$, then $\mathcal{Y}$ is of the form $l \wedge \mathcal{Y}$, where $(l, T') \in T \setminus (T_1 \cup T_2)$ and $\mathcal{Y}' \in \mathcal{C}(T')$. By construction $m_1 <_\pi l$, hence for all $l' \in \mathcal{Y}$, $m_1 <_\pi l'$ and it is impossible to have $\mathcal{X} \subseteq \mathcal{Y}$.

Right-in-Left Inclusion. To prove the right-in-left inclusion, we consider a constraint $\mathcal{Y} = l \vee \mathcal{Y}' \in \mathcal{C}(T)$ where $(l, T') \in T$ and $\mathcal{Y}' \in \mathcal{C}(T')$ such that $\mathcal{X} \nsubseteq \mathcal{Y}$. Again, the base cases are covered in the first paragraph of the proof. If $\mathcal{X} = \top$ then $\mathcal{X} \subseteq \mathcal{Y}$, a contradiction. Thus we can define $m_1 = \min_{<_\pi} \{m \in \mathcal{X}\}$ and distinguish three cases.

- If $l = m_1$ then $\mathcal{X} \setminus \{m_1\} \nsubseteq \mathcal{Y}'$ and $(l, T') \in T_1$, thus by the induction hypothesis $\mathcal{Y}' \in \mathcal{C}(T'_{out})$ where $T'_{out} = \textsc{pruneIncluded}(\mathcal{X} \setminus \{m_1\}, T')$ thus $(l, T'_{out}) \in T_{out1} \subseteq T_{out}$.

- If $l <_\pi m_1$ then $(l, T') \in T_2$. Since $\mathcal{X} \nsubseteq \mathcal{Y}'$, by the induction hypothesis $\mathcal{Y}' \in \mathcal{C}(T'_{out})$ where $T'_{out} = \textsc{pruneIncluded}(\mathcal{X}', T')$. Thus $(l, T'_{out}) \in T_{out2} \subseteq T_{out}$.

- If $m_1 <_\pi l$ then $(l, T') \in (T \setminus (T_1 \cup T_2)) \subseteq T_{out}$.

In all three cases, $\mathcal{Y} \in \mathcal{C}(T_{out})$. ∎

Algorithm 4 $\textsc{pruneInf}([C \,|\, \mathcal{X}], T, N)$

Require: $N.T$ is a normalized c-tree, $[C \,|\, \mathcal{X}]$ is a normalized c-clause.
Ensure: $\mathcal{C}_c(T_{out}) = \{[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T) | (C \npreceq D \vee N) \;\vee\; \mathcal{X} \nsubseteq \mathcal{Y}\}$,
 where $T_{out} = \textsc{pruneInf}([C \,|\, \mathcal{X}], T, N)$.

$T_{out1} \leftarrow \{(l, \textsc{pruneInf}([C \,|\, \mathcal{X}], T', N \vee l)) \,|\, (l, T') \in T$
  $\wedge\, \textsc{pruneInf}([C \,|\, \mathcal{X}], T', N \vee l) \neq \emptyset\}$
if $C \npreceq N$ **then**
  return $T_{out1} \cup \{(\boxdot, T') \,|\, (\boxdot, T') \in T\}$
else
  return $T_{out1} \cup \{(\boxdot, \textsc{pruneIncluded}(\mathcal{X}, T')) \,|\, (\boxdot, T') \in T$
    $\wedge\, \textsc{pruneIncluded}(\mathcal{X}, T') \neq \emptyset\}$
end if

Proposition 5.19 *Let $[C \,|\, \mathcal{X}]$ be a normalized c-clause, and $N.T$ be a normalized c-tree. The call $\textsc{pruneInf}([C \,|\, \mathcal{X}], T, N)$ terminates and the output c-tree $T_{out} = \textsc{pruneInf}([C \,|\, \mathcal{X}], T, N)$ is such that:*

$$\mathcal{C}_c(T_{out}) = \{[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T) \,|\, (C \npreceq D \vee N) \vee \mathcal{X} \nsubseteq \mathcal{Y}\}$$

Proof. We proceed by double inclusion and induction. Let $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$.

- Assume first that $D = \square$. If $C \not\preceq N$ then T_{out} is returned at Line 3 thus $[\square \mid \mathcal{Y}] \in \mathcal{C}_c(\{(\boxdot, T')\}) \subseteq \mathcal{C}_c(T)$ and $C \not\preceq N \vee D$. Otherwise T_{out} is generated at Line 5 and $[\square \,|\, \mathcal{Y}] \in \mathcal{C}_c(\{(\boxdot, \textsc{pruneIncluded}(\mathcal{X}, T'))\})$, with $(\boxdot, T') \in T$. By Proposition 5.18, $\mathcal{Y} \in \mathcal{C}(T')$ and $\mathcal{X} \not\subseteq \mathcal{Y}$, hence the result.

- Otherwise D is of the form $l \vee D'$ where $l = \min_{<_\pi}(D)$ and $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out1})$, thus $[D' \mid \mathcal{Y}] \in \mathcal{C}_c(\textsc{pruneInf}([C \mid \mathcal{X}], T', N \vee l))$ for some $(l, T') \in T$. By the induction hypothesis, we have $[D' \,|\, \mathcal{Y}] \in \mathcal{C}_c(T')$ (hence $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T)$) and either $C \not\preceq D' \vee N \vee l$, i.e. $C \not\preceq D \vee N$, or $\mathcal{X} \not\subseteq \mathcal{Y}$.

Now consider $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T)$ such that either $C \not\preceq D \vee N$ or $\mathcal{X} \not\subseteq \mathcal{Y}$.

- If $D = \square$ then $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(\{(\boxdot, T')\}) \subseteq \mathcal{C}_c(T)$. If $C \preceq N$ then Line 5 is reached and by hypothesis $\mathcal{X} \not\subseteq \mathcal{Y}$, thus $\mathcal{Y} \in \textsc{pruneIncluded}(\mathcal{X}, T')$ by Proposition 5.18, ensuring that $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$. Otherwise Line 3 is reached and again $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$.

- Otherwise D is of the form $l \vee D'$, where $l = \min_{<_\pi}(D)$ and $(l, T') \in T$. By hypothesis, either $C \not\preceq D \vee N$ in which case $C \not\preceq D' \vee N \vee l$, or $\mathcal{X} \not\subseteq \mathcal{Y}$. In both cases by the induction hypothesis $[D' \mid \mathcal{Y}] \in \mathcal{C}_c(\textsc{pruneInf}([C \mid \mathcal{X}], T', N \vee l))$ thus $[D \mid \mathcal{Y}] \in \mathcal{C}_c(T_{out1}) \subseteq \mathcal{C}_c(T_{out})$. ■

Theorem 5.20 *Let $N.T$ be a normalized c-tree and let $M \vee C$ and N be normalized clauses. If $M \models N$, $m \not\models N$ for all $m \in C^+$ and* $\textsc{isEntailed}([C \vee M \,|\, \mathcal{X}],\ N.T,\ \square,\ \square) =$ ***false***, *then*

$$\mathcal{C}_c(T_{out}) = \{[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T) \,|\, [C \vee M \,|\, \mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N \,|\, \mathcal{Y}]\}\,,$$

where $T_{out} = \textsc{pruneEntailed}([C \,|\, \mathcal{X}],\ T,\ M,\ N)$.

Proof. The termination of this algorithm is ensured because for all recursive calls, the value of $|C| + \mathrm{size}(T)$ strictly decreases.

Left-in-Right Inclusion. We first prove that $\mathcal{C}_c(T_{out}) \subseteq \{[D \mid \mathcal{Y}] \in \mathcal{C}_c(T) | [C \vee M \mid \mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N \,|\, \mathcal{Y}]\}$. We consider a c-clause $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$ and distinguish several cases.

- If $C = \square$ then T_{out} is generated at Line 2. By Proposition 5.19, either $M \not\preceq D \vee N$ or $\mathcal{X} \not\subseteq \mathcal{Y}$, thus $[M \,|\, \mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N \,|\, \mathcal{Y}]$ and we have the result.

- Otherwise C is of the form $m_1 \vee C'$, where m_1 is such that $m_{1\lfloor N} = \min\limits_{<_\pi} \{m_{\lfloor N} \mid m \in C\}$.

  - If $m_{1\lfloor N}$ is a contradiction, then T_{out} is set to $\textsc{pruneEntailed}([C \backslash m_1 \,|\, \mathcal{X}],\ T,\ M \vee m_1,\ N)$ at Line 6, and $m_1 \models N$ by Theorem 3.10. Since m_1 is negative, $(C \setminus m_1)^+ = C^+$ thus the preconditions of the recursive call are straightforwardly verified. By the induction hypothesis $[(C \setminus m_1) \vee m_1 \vee M \,|\, \mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N \,|\, \mathcal{Y}]$, thus $[C \vee M \,|\, \mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N \,|\, \mathcal{Y}]$.
  - Otherwise, respectively by Theorem 3.10 and the hypothesis, $m_1 \not\models N$ both when m_1 is negative and positive, hence $C \vee M \not\models N$. If $D = \square$, then $[D \mid \mathcal{Y}] \in \mathcal{C}_c(T_\boxdot)$, and $[C \vee M \mid \mathcal{X}] \not\leq_{\mathrm{C}} [N \,|\, \mathcal{Y}]$. Otherwise D is of the form $l \vee D'$, where $(l, T'_{out}) \in T_{out}$ and $[D' \,|\, \mathcal{Y}] \in \mathcal{C}_c(T'_{out})$, and one of the following holds:

Algorithm 5 pruneEntailed($[C\,|\,\mathcal{X}]$, T, M, N)

Require: $N.T$ is a normalized c-tree, $M \vee C$ is a normalized non-tautological clause, $M \models N$, $m \not\models N$ for all literals $m \in C^+$ and isEntailed($[C \vee M\,|\,\mathcal{X}]$, $N.T$, $\Box$, $\Box$) = **false**.
Ensure: $\mathcal{C}_c(T_{out}) = \{[D\,|\,\mathcal{Y}] \in \mathcal{C}_c(T) \mid [C \vee M\,|\,\mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N\,|\,\mathcal{Y}]\}$,
where T_{out} = pruneEntailed($[C\,|\,\mathcal{X}]$, T, M, N).

1: **if** $C = \Box$ **then**
2:     **return** pruneInf($[M\,|\,\mathcal{X}]$, T, N)
3: **end if**
4: let $m_1 \in C$ such that $m_{1\lfloor N} = \min_{<_\pi} \{m_{\lfloor N} \mid m \in C\}$
5: **if** $m_{1\lfloor N}$ is a contradiction **then**
6:     **return** pruneEntailed($[C \setminus m_1\,|\,\mathcal{X}]$, T, $M \vee m_1$, N)
7: **end if**
8: $T_{\boxdot} \leftarrow \{(\boxdot, T') \in T\}$
9: $T_1 \leftarrow \big\{(l, T') \in T \mid l \text{ is negative} \wedge m_{1\lfloor N} \succeq l\big\}$
10: $T_{out1} \leftarrow \{(l,$ pruneEntailed($[C\,|\,\mathcal{X}]$, T', M, $N \vee l$)$)\,|$
$(l, T') \in T_1 \wedge$ pruneEntailed($[C\,|\,\mathcal{X}]$, T', M, $N \vee l$) $\neq \emptyset\}$
11: **if** m_1 is positive **then**
12:     $T_2 \leftarrow T \setminus (T_1 \cup T_{\boxdot})$
13:     $T_{out2} \leftarrow \{(l,$ pruneEntailed($[C \setminus L_l\,|\,\mathcal{X}]$, T', $M \vee L_l$, $N \vee l$)$)\,|$
$(l, T') \in T_2 \wedge L_l = \{m \in C \mid l_{\lfloor N \vee m^c} \text{ is tautological}\} \wedge$
pruneEntailed($[C \setminus L_l\,|\,\mathcal{X}]$, T', $M \vee L_l$, $N \vee l$) $\neq \emptyset\}$
14:     **return** $T_{out1} \cup T_{out2} \cup T_{\boxdot}$
15: **else**
16:     **return** $T_{out1} \cup (T \setminus T_1)$
17: **end if**

1. $(l, T'_{out}) \in T_{out1}$, in which case there exists $(l, T') \in T$ such that l is negative, $m_{1\lfloor N} \succeq l$ and T'_{out} = pruneEntailed($C, T', M, N \vee l$). Since $M \models N$ by hypothesis, we have $M \models N \vee l$. Furthermore, since $N.T$ is a c-tree and l occurs in T, we must have $\forall l' \in N$, $l' <_\pi l$, hence, since l is negative, so is $N \vee l$ by definition of the ordering $<_\pi$. Thus for all $m \in C^+$, $m \not\models N \vee l$. Now $(N \vee l).T'$ is a normalized c-tree because $N.T$ is normalized, hence the preconditions of the recursive call at Line 10 are all met, and $[C \vee M \mid \mathcal{X}] \not\leq_{\mathrm{C}} [D' \vee l \vee N\,|\,\mathcal{Y}] = [D \vee N\,|\,\mathcal{Y}]$ by the induction hypothesis.

2. $(l, T'_{out}) \in T_{out2}$, in which case m_1 is a positive literal. By setting $L_l = \{m \in C \mid l_{\lfloor N \vee m^c} \text{ is tautological}\}$, we then have T'_{out} = pruneEntailed($C \setminus L_l, T', M \vee L_l, N \vee l$). By Theorem 3.10, $M \vee L_l \models N \vee l$. If there exists an $m \in (C \setminus L_l)^+$ such that $m \models N \vee l$, then $l_{\lfloor N \vee m^c}$ is a tautology because $m \not\models N$ by the precondition of the algorithm. But by definition, such an m should belong to L_l and we have a contradiction. Thus, the preconditions of the recursive call at Line 13 are verified and by the induction hypothesis, $[M \vee L_l \vee (C \setminus L_l)\,|\,\mathcal{X}] \not\leq_{\mathrm{C}} [D' \vee l \vee N\,|\,\mathcal{Y}]$, i.e., $[M \vee C\,|\,\mathcal{X}] \not\leq_{\mathrm{C}} [D\,|\,\mathcal{Y}]$.

3. $(l, T'_{out}) \in T \setminus T_{out1}$ and T_{out2} is not computed, in which case m_1 is negative and $m_{1\lfloor N} \prec l$. By Proposition 3.17 and Theorem 3.10, $m_1 \not\models D$ thus $C \not\models N \vee D$.

Right-in-Left Inclusion. For the converse inclusion, let $[D \,|\, \mathcal{Y}]$ be a clause in $\mathcal{C}_c(T)$ such that $[C \vee M \,|\, \mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N \,|\, \mathcal{Y}]$.

- If $C = \square$, then since $M \models N$, either $M \not\preceq D \vee N$ or $\mathcal{X} \not\subseteq \mathcal{Y}$. The tree T_{out} must be generated at Line 2 and, by Proposition 5.19, $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$.
- Otherwise C is of the form $m_1 \vee C'$ where m_1 is such that $m_{1\lfloor N} = \min\limits_{<_\pi} \{m_{\lfloor N} \mid m \in C\}$.
  - If $m_{1\lfloor N}$ is a contradiction then $m_1 \models N$ by Theorem 3.10 and T_{out} is returned at Line 6. By the induction hypothesis $[C \setminus m_1 \vee m_1 \vee M \,|\, \mathcal{X}] \not\leq_{\mathrm{C}} [D \vee N \,|\, \mathcal{Y}]$, and therefore $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$.
  - Otherwise we may have either $D = \square$, in which case $[D \mid \mathcal{Y}] \in \mathcal{C}_c(T_\boxdot)$ and the result is straightforward, or D is of the form $l \vee D'$ for some $(l, T') \in T$ and $[D' \,|\, \mathcal{Y}] \in \mathcal{C}_c(T')$.
    1. If l is negative and $m_{1\lfloor N} \succeq l$, then $(l, T') \in T_1$. In addition, T'_{out1} is set to pruneEntailed$([C \,|\, \mathcal{X}], T', M, N \vee l)$ at Line 10. The preconditions of the recursive call are all met for the same reason as in the first inclusion proof. By the induction hypothesis $[D' \,|\, \mathcal{Y}] \in \mathcal{C}_c(T'_{out1})$ thus $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out1})$.
    2. If m_1 is negative and either l is positive or $m_{1\lfloor N} \prec l$ then $(l, T') \in T \setminus T_1$ and T_{out} is set at Line 16, thus $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out})$.
    3. If the previous conditions do not hold, then m_1 is a positive literal. By Theorem 3.10, the clause $L_l = \{m \in C | l_{\lfloor N \vee m^c}$ is tautological$\}$ is such that $L_l \models N \vee l$, and since $M \models N$, we have $M \vee L_l \models N \vee l$. Moreover $(N \vee l).T'$ is a normalized c-tree and for all m occuring in $(C \setminus L_l)^+$, $m \not\models N \vee l$ for the same reason as in the previous inclusion. Hence the preconditions of the recursive call pruneEntailed$([C \setminus L_l \,|\, \mathcal{X}], T', M \vee L_l, N \vee l)$ are verified. Since $[C \setminus L_l \vee M \vee L_l \,|\, \mathcal{X}] \not\leq_{\mathrm{C}} [D' \vee l \vee N \,|\, \mathcal{Y}]$, by the induction hypothesis, $[D' \,|\, \mathcal{Y}] \in \mathcal{C}_c(T'_{out2})$ and $[D \,|\, \mathcal{Y}] \in \mathcal{C}_c(T_{out2})$. ■

Remark 5.21 *Note that sets of implicates generated by the main saturation algorithm (i.e., the clauses C such that the final c-tree contains a c-clause $[\square \mid \mathcal{X}]$ with $\mathcal{X}^{\mathrm{c}} = C$) can be eventually stored in c-trees with empty constraints. In this case, the $\preceq$-condition can be omitted when testing redundancy, since the goal is to remove all implicates that are redundant w.r.t. logical entailment. The obtained algorithms are very similar to (and simpler than)* isEntailed *and* pruneEntailed. *They are omitted for the sake of conciseness.*

6. Experimental Results

In this section, we present the different tools and benchmarks used to conduct experiments, as well as the results obtained from those experiments.

6.1 Implementation

Our prime implicate generation method has been implemented in a research prototype written in OCaml[7]. The system, called `cSP`, is available at `https://forge.imag.fr/docman/?group_id=683`. The input formulæ are sets of equational ground clauses in the TPTP syntax v5.4.0.0 (Sutcliffe, 2009). Another version of the program, called `cSP_flat`, implements additional refinements that only apply to *flat* clauses, i.e., to clauses not containing function symbols of an arity greater than 0. The system for non-flat clauses is based on the `LogTk` library (Cruanes, 2014) for the handling of term ordering and congruence closure.

There are several options available for controlling the form of the generated implicates: `-max-size`, `-max-neg` and `-max-depth` limit the length of clause, the number of negative literals and the depth of the terms, respectively; `-cov` accepts only implicates that entail one of the clauses of the input formula (for generating a minimal cover of the input formula). To ensure termination, a restriction must be added to the terms introduced by the Assertion rules, since otherwise the rules are infinitely branching, see Section 4. By default, `cSP` considers all the terms occurring in the initial formula. A finite set of terms can also be provided by the user in a TPTP input file with the extension '.conf'.

The order in which the c-clauses are processed is fixed by comparing the size of the constraints and of the clausal parts, in lexicographic increasing order. A more detailed description of the system was done by Tourret (2016).

6.2 Experimental Context

This section describes the reference tools and of the benchmarks used in the experiments.

6.2.1 Reference Tools.

Tools for the computation of prime implicates in propositional and first order logic directly available from the web include only (to the best of our knowledge):

`ritrie` (Matusiewicz, Murray, & Rosenthal, 2009), a propositional tool that specializes in fast querying of implicates[8].

`primer` (Previti, Ignatiev, Morgado, & Marques-Silva, 2015), a prime implicate generation tool based on satisfiability encoding[9].

`Mistral` (Dillig & Dillig, 2013), an SMT solver which can be used for performing abductive inferences[10].

Thanks to the kindness of their respective authors, we also obtained two other tools:

`Zres` (Simon & Del Val, 2001), an implicate generation tool in propositional logic based on Resolution,

`SOLAR` (Iwanuma, Nabeshima, & Inoue, 2009), a prime implicate generation tool for first order logic, including equational logic, based on Semantic Tableaux.

7. `http://ocaml.org`
8. `http://www.cs.albany.edu/ritries/prototype.html`
9. `http://logos.ucd.ie/web/doku.php?id=primer`
10. `http://www.cs.utexas.edu/~tdillig/mistral/index.html`

We compared `cSP` with all these systems except for `ritrie` and `Mistral`. The former is because it was not efficient enough, which can be explained by the fact that it was built to perform efficient querying of an already generated set of prime implicates, rather than to compute efficiently the said set (which it can nevertheless do). The latter is because it cannot be compared with our work, because its prime implicate generation is not complete.

We compared the systems on three sets of test problems, two of which are randomly generated due to the lack of existing benchmark in the targeted logics. These benchmarks are included in the archive containing the source code of `cSP`. In each case, the propositional equivalents of the problems were obtained by instantiating the transitivity axioms for all constant symbols appearing in them – the reflexivity and commutativity axioms are encoded directly in the transformation by orienting and simplifying the equations. Thus the propositional tools have to handle much bigger inputs and they generate many more prime implicates than their counterparts handling more expressive logics (for instance they also compute the implicates of the transitivity instances). To recover the same results as the other tools, it would be necessary to replace the propositional variables by the corresponding equations and to remove all redundancies. This step is not performed here since it is straightforward (although costly), but in an application case, it would have to be performed to reach an intelligible result. Similarly, non-flat clause sets can also be flattened by adding all relevant instances of the substitutivity axioms and unflattened in a post-processing step that is omitted here.

6.2.2 Flat Random Benchmarks.

As far as we are aware, there are no benchmarks for ground flat (i.e., without function symbols of an arity strictly greater than 0) equational logic. Our attempts with flattened ground problems from the TPTP library did not succeed, because none of the available tools can generate all the prime implicates of such large formulæ with reasonable time and memory constraints. We thus created our own benchmark, made of a thousand randomly generated formulæ of 6 clauses build on 8 constants and containing between 1 and 5 literals (even such small formulæ can have very large set of prime implicates).

6.2.3 Non-Flat Random Benchmarks.

The formulæ in this benchmark contain function symbols and were generated using the following parameters:

- $\mathtt{a} \in \{1, 2\}$, is the maximal arity of the functions and $\mathtt{a} * 5$ is the size of the randomly generated signature[11],
- $\mathtt{c} \in \{2, 3, 4\}$ is the number of clauses in a formula,
- $\mathtt{d} \in \{1, 2\}$ is the maximal depth of the terms,
- $\mathtt{l} \in \{2, 3\}$ is the maximal number of literals in a clause (the minimum being 1),

The formulæ are stored in files that are named with the following convention: `sa_d_c_l` (e.g. `s1_2_4_2`). In total, this benchmark contains 144 formulæ.

11. By design, the signature contains at least two literals of arity zero.

6.2.4 Other Benchmarks.

These benchmarks are extracted from the SMT-LIB database (Barrett, Fontaine, & Tinelli, 2015) in the logic QF_AX, i.e. "closed quantifier-free formulas over the theory of arrays with extensionality"[12]. These are synthetic benchmarks that model some properties of arrays with extensionality, namely:

- the order in which the elements are stored in an array does not matter (`storecomm` benchmarks),
- some swappings of elements between cells of an array are commutative (`swap` benchmarks),
- swapping elements between identical cells of equal arrays generates equal arrays (`storeinv` benchmarks)

The creators of these benchmarks slightly altered them to falsify these properties, thus creating the benchmarks labeled with `invalid`[13]. Given that `cSP` cannot handle SMT-LIB inputs or the theory of arrays, we preprocessed the benchmarks by first converting them to TPTP and then applying the algorithm described by Bonacina and Echenim (2010). As shown by Armando et al. (2009), these problems can be nontrivial to solve even for state-of-the-art theorem provers like `E` (Schulz, 2013) and one cannot expect the entire set of prime implicates to be generated in reasonable time. We use them mainly to evaluate the impact of our redundancy-pruning technique on the number of Superposition inferences carried out by blocking the Assertion rules inferences (i.e. applying a filter blocking all implicates apart from the empty one), allowing the comparison of `cSP` with the `E` theorem prover.

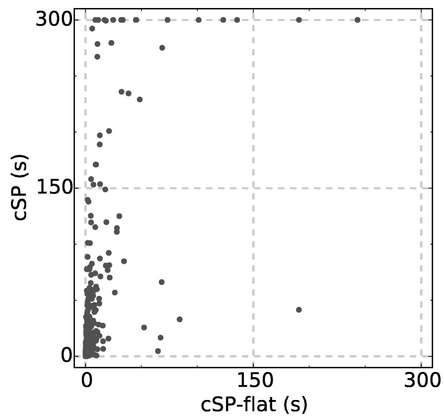
6.3 Results

All of the experiments were run on a machine equipped with an Intel core i5-3470 CPU and 4×2 GB of RAM running Ubuntu 14.04. In a first experiment we compare `cSP` with `cSP_flat` and `primer` on the random flat benchmark to observe the impact of the functional term representation (`LogTk`) of `cSP`. Then we compare our tools to state-of-the-art solvers on the non-flat benchmark. Finally, we observe the behavior of `cSP` on bigger formulæ.

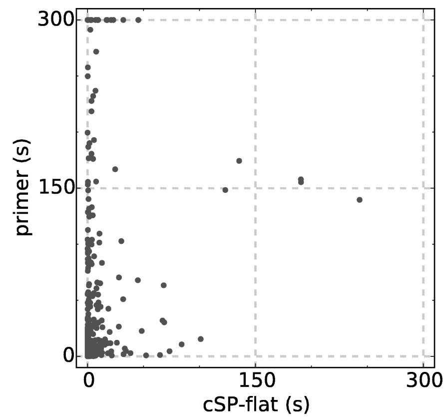
6.3.1 Impact of the Handling Functional Terms.

Handling functional terms is costly. This can be seen on Figure 3, that compares the execution time of `cSP` on the random flat benchmark with that of `cSP_flat` and `primer`. As shown on Figure 3a, on average, `cSP` is ten times slower than `cSP_flat` on the random flat benchmark. This overhead induced by functional terms is easily explainable: the new term representation must handle nested terms, which prevents the use of a simple integer representation as is done in `cSP_flat`. Moreover, the subsumption method is a bit more involved which plays a role in slowing down `cSP` compared to `cSP_flat`. Figure 3b compares the execution time of `cSP_flat` with that of `primer` on the same benchmark to illustrate the performance of `cSP_flat` in ground flat equational logic.

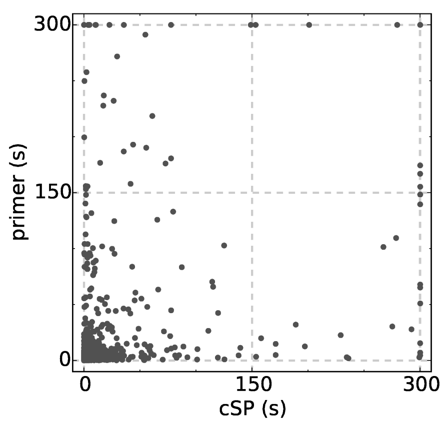
12. `http://smtlib.cs.uiowa.edu/logics.shtml`
13. For details on these benchmarks, we refer the interested reader to the paper by Armando, Bonacina, Ranise, and Schulz (2009).



(a) cSP vs. cSP_flat



(b) primer vs. cSP_flat



(c) primer vs. cSP

	slower	equal	faster
cSP_flat	12.3%	0%	87.7%
cSP	39.1%	0.4%	60.5%

(d) Proportion of formulæ that are slower/equal/faster on cSP_flat/cSP than on primer

Figure 3: Time comparison of cSP, cSP_flat and primer; random flat benchmark

6.3.2 Performance Comparisons on Random Non-Flat Benchmark.

The experiment presented below is a comparison of the prime implicate generation systems Zres, primer and SOLAR with cSP_flat and cSP on the random non-flat benchmark.

The results are summarized in Table 4. Each line corresponds to a system. The column labeled 'successes' indicates the percentage of tests that were completed before the 5 minute timeout. The four columns under the label 'SOLAR successes' summarize average results on those tests on which SOLAR terminated before the timeout. The same goes for the columns under 'Zres successes', 'primer successes', 'cSP_flat successes' and 'cSP successes'. Finally, the 'timeout' columns expose the mean results on all interrupted tests. Columns labeled 'fail', 'time', 'inf.' and 'PIs' respectively give the percentage of formulæ on which the system timed out relative to this part of the benchmark, the mean execution time, mean number of

	successes	SOLAR successes				Zres successes			
		fail.	time(s)	inf.	PIs	fail.	time(s)	inf.	PIs
SOLAR	15%	0%	11.842	663190	506	74%	13.608	767160	455
Zres	52%	13%	0.695	X	2986	0%	12.474	X	13804
primer	53%	13%	0.794	X	2986	0%	3.770	X	13847
cSP_flat	63%	4%	6.622	2275	74	2%	0.500	1737	159
cSP	76%	0%	0.042	99	21	2%	3.576	755	49

	primer successes				cSP_flat successes			
	fail.	time(s)	inf.	PIs	fail.	time(s)	inf.	PIs
SOLAR	75%	13.608	767160	455	76%	12.360	694523	460
Zres	2%	12.474	X	13804	19%	7.073	X	10405
primer	0%	8.016	X	18949	17%	7.590	X	15779
cSP_flat	2%	1.480	2347	180	0%	14.290	6730	348
cSP	2%	4.296	851	49	3%	7.556	1004	62

	cSP successes				timeouts	
	fail.	time(s)	inf.	PIs	inf.	PIs?
SOLAR	79%	11.842	663191	506	2452908	28152
Zres	33%	10.391	X	11338	X	X
primer	31%	7.671	X	16687	X	X
cSP_flat	19%	8.795	5418	313	X	X
cSP	0%	10.193	1209	79	14714	538

Table 4: Test results summary; random non-flat benchmark

	slower	equal	faster
cSP_flat	5.6%	41.7%	52.8%
cSP	23.6%	3.5%	72.9%

Table 5: Proportion of formulæ that are slower/equal/faster on cSP_flat/cSP than on primer

inferences and mean number of prime implicates found for each set of tests. The last column under the 'timeout' label is labeled 'PIs?' because, due to the timeout, the implicates found are not guaranteed to be prime. Cells labeled with an 'X' indicate that the corresponding data is not available.

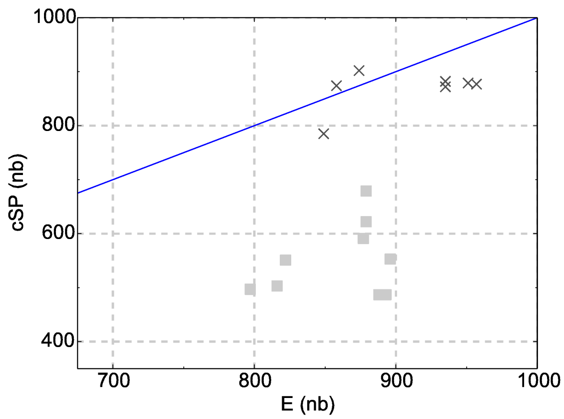
As shown in the 'successes' column, cSP is the obvious winner in terms of the number of tests handled before timeout. The 15% of problems solved by SOLAR are the simplest of the random formulæ. The results show that SOLAR's approach is very costly both in terms of time and space. The high number of prime implicates this tool generates compared to those produced by cSP is due to the fact that SOLAR does not take into account the equality axioms in its redundancy detection. Thus for example, any literal $t \simeq s$ also appears as $s \simeq t$, and $f(s) \simeq f(t)$ is not detected as redundant w.r.t. $s \simeq t$. The huge number of prime implicates generated by Zres and primer stems directly from the lack of post-processing (conversion of the results back to equational logic)[14]. This prevents the detection of the purely equational redundancies, e.g. clauses redundant w.r.t. the transitivity axiom. Although Zres and primer are faster than cSP on the problems they both solve, they solve 52% and 53% of the benchmark respectively, while cSP solves 76% of them. The results in the 'cSP successes' and 'cSP_flat successes' columns are globally higher than those in the 'Zres successes' and 'primer successes' columns, because the most difficult formulæ are solved only by cSP and to a lesser extend by cSP_flat. Since cSP solves more problems than cSP_flat and does so faster and with fewer clauses processed, cSP is clearly better adapted to dealing with originally non-flat formulæ. Incidentally, note that the overhead of cSP's term handling compared to that of cSP_flat's (observed in the previous experiment) is more than compensated by the direct handling of non-flat formulæ since on the random non-flat benchmark cSP is faster than cSP_flat. The number of inferences and generated non-redundant implicates when the tool times out illustrates the heavy cost of the cSP inferences and redundancy detection mechanism compared to that of SOLAR. It is a price that seems partly unavoidable to eliminate all redundancies, since this requires costly algorithms.

6.3.3 Impact of Normalization.

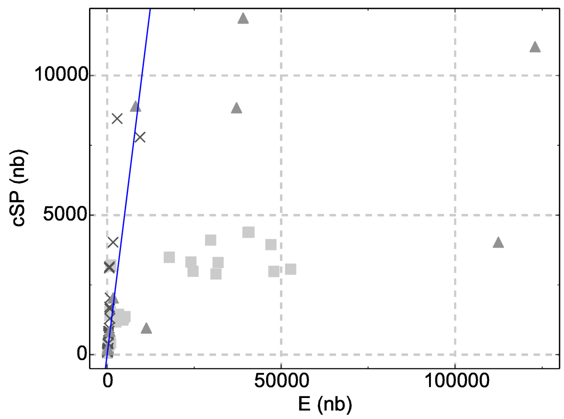
We also assessed the impact of the normalization mechanism on the QF-AX benchmark. This was done by comparing cSP with the E theorem prover, running cSP with a filtering option (-max-size 0) to block the generation of any implicate besides the empty clause, effectively turning cSP into a Superposition theorem prover. This way the main differences between cSP and E are the normalization of clauses and the redundancy pruning mechanism. On the one hand, the redundancy pruning algorithm used by cSP is weaker because it does not allow for equational simplification or other n-to-one redundancy pruning rules. On the other hand, one-to-one redundancy testing is stronger since its uses logical entailment together with the usual ordering condition instead of subsumption. The comparison of cSP with the E theorem prover on these formulæ shows that the normalization approach can, in some nontrivial cases, reduce the number of processed clauses by an order of magnitude.

Figure 4 presents the positive results of this experiment, i.e., the results of the storecomm and swap formulæ. Among these, only the formulæ on which both E and cSP (without Assertion rules) terminate before the timeout and without memory overflow were kept. Light gray squares represent the invalid formulæ, i.e. the satisfiable ones, while dark gray crosses mark the unsatisfiable ones. The line $y = x$ is added in both plots. An interesting observation is that for the largest invalid formulæ, cSP needs to process a smaller number of clauses than E before terminating, even 10 times less in the case of the invalid_swap formulæ. In addition, the unsatisfiable swap formulæ were run with a timeout of 10 minutes (the triangles in Plot (4b)) and the corresponding results hint that this phenomenon could also be true for larger unsatisfiable problems. This suggests that the redundancy pruning technique based on normalization and clausal trees could be profitably integrated into state-of-the-art Superposition-based theorem-provers, at least for ground equational clause sets.

14. It should be noted that the mean number of prime implicates generated by Zres and primer sometimes differ a lot. This is due to a bug in Zres which sometimes makes it incomplete.



(a) `storecomm` formulæ



(b) `swap` formulæ

Figure 4: Comparison of the number of processed clauses for `E` and `cSP`.

However, it might not always be useful, e.g. Figure 5, plotting the `storeinv` formulæ where an opposite tendency is observed, although the `storeinv` formulæ on which neither `cSP` or `E` timeout represent only a tenth of that of the other two families, for which the results are positive.

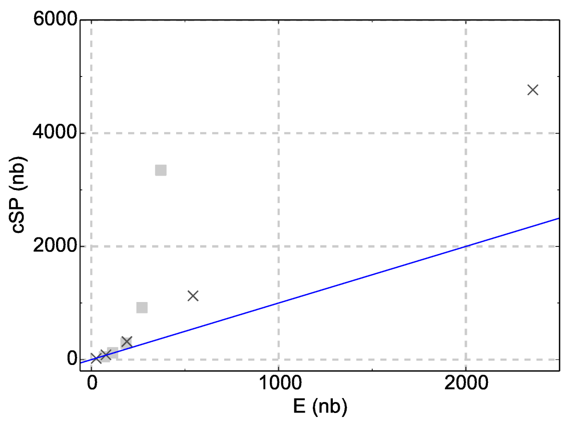


Figure 5: `storeinv` formulæ - comparison of the number of processed clauses for `E` and `cSP`.

7. Discussion

In this section, we present potential applications of our work in AI, discuss related work, and provide some lines of future research.

7.1 Summary

The present work comprises three main contributions:

- The c$\mathcal{SP}$ calculus extends the classical Superposition calculus for first-order logic with equality with two rules that allow the addition of abductive hypotheses to the generated clauses.
- A clause storage data structure is defined, the *constrained clausal tree*. It significantly reduces memory consumption and allows the efficient detection and removal of redundant clauses.
- Two prototypes of prime implicate generation tools implement the calculi, storage method and their variants, one that handles only ground flat terms and one that handles arbitrary ground terms.

Our main results are the following. On the theoretical side, we proved the deductive-completeness of the calculus for full equational logic and the termination and correctness of the constrained clausal tree manipulation algorithms for ground equational logic. On the practical side we conducted an experimental evaluation of our prototypes (on flat and non-flat benchmarks). The different experiments presented in Section 6 allow us to draw the following picture of our prime implicate generation algorithm:

1. `cSP` compares favorably with state-of-the-art prime implicate generation tools. As indicated by the results in Table 4, `cSP` solves more problems (and more efficiently) than all others available tools. Precisely, our best prototype is faster than the reference tool (`primer`) in 87.7% of the benchmark in flat ground logic and 72.9% in non-flat ground logic. However, none of the systems scale well: the formulæ for which complete sets of prime implicates can be effectively computed are rather simple.
2. The normalization method used in our algorithms significantly reduces the number of generated clauses compared to the state-of-the-art `E` theorem prover (although our inference engine is far less efficient). This suggests that the integration of this technique to Superposition-based theorem-provers could be beneficial.
3. Our experiments also suggest that it is always better to have a method tailored to the input formulæ rather than to preprocess them by flattening and/or conversion to propositional logic or to use a tool for a more expressive logic. For example, on flat formulæ, `cSP_flat` is the best choice over `cSP`, while on non-flat ones `cSP` gives the best results.

7.2 Applications in Artificial Intelligence

When they terminate, usual proof procedures in first-order logic return a yes or no result, together with a formal justification: either the input formula is valid, and in this case a proof is provided; or it is not, and in some cases a counter-example can be constructed. In contrast, the generation of prime implicates allows for "open" requests. It thus targets situations when the possible answers are not known, but have to be constructed by the program, as well as their justification. This makes the reasoning task much more complex and significantly increases the search space, but also allows for a wider range of applications, and for the conception of more "intelligent" programs, able to derive results that are not necessarily expected by human users. So far the implicate generation problem has been tackled mostly for propositional and related logics, although many applications require more expressive logics for modeling systems or knowledge bases. Finding implicate generation algorithms that are capable of handling expressive logics is thus a relevant and important problem in Artificial Intelligence. In particular, being able to handle equalities efficiently is a prerequisite, since adding equality axioms would be very inefficient. For instance the OWL Web Ontology Language (Patel-Schneider, Hayes, Horrocks, et al., 2004) includes atoms $\texttt{SameIndividual}(a_1, \dots, a_n)$ that assert that all of the individuals a_i (for $i \in [1, n]$) are equal to each other. We provide below some concrete examples of applications.

7.2.1 Diagnosis

Diagnosis, defined as the design of techniques to determine whether the behavior of a system is correct (i.e., fulfills its specification) and to generate suitable explanations if it is not, is usually viewed as an important subdomain of AI. We target model-based diagnosis, in which the behavior of the system and its properties are modeled by logical formulas. For many applications, propositional logic is not expressive enough, and richer logics must be used instead: for instance, many verification problems are based on ground equational logic or can be reduced to it (see, e.g., Cimatti, Griggio, & Sebastiani, 2011). The generation of prime implicates is useful in this context for explaining bad or unexpected behaviors of a system. The idea is to determine what the additional (if possible minimal) hypotheses that could be added to enforce the desired properties are. This allows one to identify missing conditions (if the additional hypotheses are fulfilled) or to give a hint of why the system does not behave as expected (if the additional hypotheses are false).

We provide an example coming from program verification. Consider a program that assigns the pointers to the tails of two lists l_1 and l_2 as follows:

$$\texttt{tail}(l_1) \leftarrow l_0; \texttt{tail}(l_2) \leftarrow l_1$$

Assume we want to check that the relation $\texttt{tail}(\texttt{tail}(l_2)) = l_0$ holds after the execution of the program. This problem can be naturally modeled in equational logic as follows. The function `tail` is represented by a function $\mathit{tail} : \textsc{heap} \times \textsc{list} \to \textsc{list}$, and the redirection operation is encoded by a function $\mathit{redir} : \textsc{heap} \times \textsc{list} \times \textsc{list} \to \textsc{heap}$, together with the following axioms:

$$\mathit{tail}(\mathit{redir}(h, x, y), x) \simeq y$$
% The tail of x is redirected to y
$$x' \not\simeq x \Rightarrow \mathit{tail}(\mathit{redir}(h, x, y), x') \simeq \mathit{tail}(h, x')$$
% The tail of the other lists is not affected by the redirection

The conclusion to be proven is

$$h'_0 \simeq \mathit{redir}(\mathit{redir}(h_0, l_1, l_0), l_2, l_1) \Rightarrow \mathit{tail}(h'_0, \mathit{tail}(h'_0, l_2)) \simeq l_0$$

where h_0 and h'_0 are constant symbols denoting the initial and final states of the heap. Trying to prove this goal from the axioms actually fails, which may come as a surprise to some users. Indeed, in the case where $l_1 = l_2$, the final list l_2 loops to itself (the tail of l_2 is l_2), and l_0 is no longer accessible. The property can be proven if l_1 is distinct from l_2, which can be inferred by generating the prime implicate $l_1 \simeq l_2$ from the specification and the negation of the conclusion.

7.2.2 Computing Possible Explanations of Observations

Similarly, abductive reasoning also allows one to construct possible explanations of observed behaviors. Given an ontology $\mathcal{O}$ and a knowledge base $\mathcal{K}$, both modeled by a set of axioms in first-order logic with equality (e.g., an OWL database), and an observation o, implicates of $\mathcal{O} \wedge \mathcal{K} \wedge \neg o$ correspond to negations of explanations of o. For example, given the axioms

$$\textit{blue-eyed}(\textit{father}(x)) \wedge \textit{blue-eyed}(\textit{mother}(x)) \Rightarrow \textit{blue-eyed}(x)$$

the facts $\textit{blue-eyed}(\textit{alice}) \wedge \textit{blue-eyed}(\textit{bob}) \wedge \textit{mother}(\textit{carol}) \simeq \textit{alice}$, and the observation $\textit{blue-eyed}(\textit{carol})$, we may infer the hypothesis: $\textit{father}(\textit{carol}) \simeq \textit{bob}$. This is done by generating the prime implicate $\textit{father}(\textit{carol}) \not\simeq \textit{bob}$ from the above axioms and the negation of the observation (assessing the actual plausibility of the assumption is out of the scope of the present work, it may depend on heuristics or other observations). Formally, this implicate is generated by the c$\mathcal{SP}$ calculus as follows (Resolution steps may be simulated by Superposition inferences and reflection).

1	$\neg\textit{blue-eyed}(\textit{father}(x)) \vee \neg\textit{blue-eyed}(\textit{mother}(x)) \vee \textit{blue-eyed}(x)$	% axiom
2	$\neg\textit{blue-eyed}(\textit{carol})$	% neg. observ.
3	$\neg\textit{blue-eyed}(\textit{father}(\textit{carol})) \vee \neg\textit{blue-eyed}(\textit{mother}(\textit{carol}))$	% Res., 1, 2
4	$\textit{mother}(\textit{carol}) \simeq \textit{alice}$	% axiom
5	$\neg\textit{blue-eyed}(\textit{father}(\textit{carol})) \vee \neg\textit{blue-eyed}(\textit{alice})$	% Sup., 3, 4
6	$\textit{blue-eyed}(\textit{alice})$	% axiom
7	$\neg\textit{blue-eyed}(\textit{father}(\textit{carol}))$	% Res., 5, 6
8	$[\neg\textit{blue-eyed}(\textit{bob}) \,\vert\, \textit{father}(\textit{carol}) \simeq \textit{bob}]$	% Ass., 7
9	$\textit{blue-eyed}(\textit{bob})$	% axiom
10	$[\square \,\vert\, \textit{father}(\textit{carol}) \simeq \textit{bob}]$	% Res., 8, 9

7.2.3 Extracting Relevant Consequences

Implicate generation can also be used to compute relevant consequences of a knowledge base, in cases where such consequences are not known in advance. Consider for instance the axiom

$$\textit{daughter}(x, y) \Leftrightarrow \textit{female}(x) \wedge y \simeq \textit{mother}(x)$$

The famous riddle "If Teresa's daughter is my daughter's mother, who am I to Teresa?" can be answered by considering the formula $\textit{daughter}(s, \textit{Me}) \wedge \textit{daughter}(\textit{mother}(s), \textit{Teresa})$ and computing some of its prime implicates:

1	$\textit{daughter}(s, \textit{Me})$	% axiom
2	$\neg\textit{daughter}(x, y) \vee y \simeq \textit{mother}(x)$	% axiom
3	$\textit{Me} = \textit{mother}(s)$	% Res., 1, 2
4	$\textit{daughter}(\textit{mother}(s), \textit{Teresa})$	% axiom
5	$\textit{Teresa} = \textit{mother}(\textit{mother}(s))$	% Res., 2, 4
6	$\textit{Teresa} = \textit{mother}(\textit{Me})$	% Sup., 5, 3
7	$[\textit{Teresa} \not\simeq \textit{Teresa} \,\vert\, \textit{mother}(\textit{Me}) \not\simeq \textit{Teresa}]$	% Ass., 6
8	$[\square \,\vert\, \textit{mother}(\textit{Me}) \not\simeq \textit{Teresa}]$	% Refl.

Clause 8 yields a first solution $mother(Me) \simeq Teresa$. But another (equivalent) solution exists:

9	$daughter(x, y) \vee \neg female(x) \vee y \not\simeq mother(x)$	% axiom
10	$[Teresa \not\simeq mother(Me) \vee \neg female(Me) \,|\, \neg daughter(Me, Teresa)]$	% Ass., 9
11	$[\neg female(Me) \,|\, \neg daughter(Me, Teresa)]$	% Res., 10, 6
12	$\neg daughter(x, y) \vee female(x)$	% axiom
13	$female(mother(s))$	% Res., 12, 4
14	$female(Me)$	% Sup., 13, 3
15	$[\Box \,|\, \neg daughter(Me, Teresa)]$	% Res., 14, 12

Note that the computed implicates should only refer to the relevant symbols Me, $Teresa$ and to the above relations, for example, they should not contain the auxiliary constant s. This motivates the fact that we should be able to compute specific prime implicates verifying user-specified criteria.

7.2.4 Dealing for Incomplete Information

Abducing equalities is also useful to deal with approximative, incomplete or even spurious information. Assume we are given an ontology $\mathcal{O}$ and knowledge base $\mathcal{K}$, and that we want to check whether some property p holds. In some cases, property p will fail to be derived, due to unreported synonyms (e.g., misspelled names, duplicate identifiers,...), aliases or missing definitions. This problem, that commonly arises in practice, may be detected by computing purely equational prime implicates of $\mathcal{O} \wedge \mathcal{K} \wedge \neg p$, effectively showing that p possibly holds provided some simple equalities are added in the knowledge base. Afterwards, one could seek confirmation from other sources and update the database if needed. For instance, given the facts:

$$mother(alice) \simeq carol \wedge mother(bob) \simeq carol$$
$$\wedge father(alice) \simeq dave \wedge father(bob) \simeq david$$

the axiom:

$$brother(x, y) \Leftrightarrow (male(x) \wedge mother(x) \simeq mother(y) \wedge father(x) \simeq father(y))$$

and the request: "who is the brother of Alice?", we may compute the plausible answer "Bob", under the assumption that $dave$ and $david$ denote the same individual. This may be done by deriving the implicate $dave \not\simeq david$ from the negation of the goal $\neg brother(x, alice)$. The variable x is unified with bob in the derivation, yielding the desired result. Similarly, if the fact $father(bob) \simeq david$ is deleted, then the above answer can still be returned, under the condition that $father(bob) \simeq dave$.

In the spirit of the previous examples, the first-order prime implicate generation tool SOLAR (Nabeshima, Iwanuma, Inoue, & Ray, 2010) is used in non-equational first-order logic to abduce hypotheses for the completion of biological networks (Rougny, Yamamoto, Nabeshima, Bourgne, Poupon, Inoue, & Froidevaux, 2015) and this technique is generalized into a method to abduce logical reasoning called meta-level abduction by Inoue (2016). Although the work presented in these papers does not rely on equational logic, its use could make meta-level abduction more powerful by giving it the possibility of unifying seemingly different objects to explain observations, which could be particularly useful when data comes from several sources, as illustrated in the previous paragraph.

7.3 Related Work

The prime implicate generation problem has been extensively investigated in the context of propositional logic (see e.g., Marquis, 2000). Standard algorithms are based mainly on refinements of the Resolution rule (Leitsch, 1997; Robinson, 1965), because unrestricted Resolution permits to generate all the prime implicates of a set of clauses (Jackson, 1992; Kean & Tsiknis, 1990; Quine, 1955; Tison, 1967). The proposed approaches then focus on the definition of efficient strategies to generate saturated clause sets, by considering literals incrementally, and on the definition of compact data structures for storing the generated sets of implicates, e.g., using tries (De Kleer, 1992; Fredkin, 1960) or Z-BDDs (Mishchenko, 2001; Simon & Del Val, 2001). Other approaches use decomposition-based methods, in the style of the DPLL procedure, for generating trie-based representations of sets of prime implicates (Matusiewicz et al., 2009; Matusiewicz, Murray, & Rosenthal, 2011). Recently, a new approach that outperforms previous algorithms has been proposed by Previti et al. (2015); it is based on satisfiability solving and problem reformulation. The idea is to associate each literal l with a variable x_l indicating whether l occurs in the considered implicate; using an adequate reformulation, the implicates correspond exactly to maximal models of formulæ (iteratively) built on these variables.

There have been only very few approaches dealing with more expressive logics. Some extensions have been considered in modal logics (see e.g., Bienvenu, 2007; Blackburn, Van Benthem, & Wolter, 2007), and algorithms have been proposed for first-order formulæ. Some of these approaches use unrestricted versions of the Resolution calculus (Knill, Cox, & Pietrzykowski, 1993; Marquis, 1991). Other procedures extend the semantic tableau method to search for hypotheses ensuring that all branches in the tableau can be closed (Mayer & Pirri, 1993; Nabeshima et al., 2010). However, none of them handle equality efficiently. More recently, algorithms were devised to generate sets of implicants of formulæ interpreted in decidable theories (Dillig, Dillig, McMillan, & Aiken, 2012), by combining quantifier-elimination for discarding useless variables, with model building to construct sufficient conditions for satisfiability. The approach does not apply to equational formulæ with (uninterpreted) function symbols since this would involve second-order quantifier elimination. For instance, generating the implicant $a \simeq b$ from $f(a) \simeq f(b)$ requires to solve the second-order quantifier problem: $\forall f\,(f(a) \simeq f(b))$ in order to get rid of the symbol f.

Every existing prime implicate generation procedure in propositional logic can be employed to handle ground equational clauses by instantiating the variables occurring in the equality axioms by the terms occurring in the input formula and considering equations as propositional variables in the augmented formula. Systems designed to handle first-order logic can also be used, simply by adding the equality axioms in the input clauses. In both cases, however, the encoding introduces many redundancies, since the axioms of equality will not be taken into account for redundancy testing. For instance, in both approaches, all logical consequences of these axioms will be generated as implicates. Furthermore, as explained in Section 3, an equational clause may possess many equivalent forms – possibly exponentially many – and every such form will be considered if the transitivity and substitutivity axioms are not taken into account[15]. Besides efficiency problems, a post-processing

15. It is clear that the commutativity and reflexivity axioms can be easy handled by the encoding, e.g., two equations $a \simeq b$ and $b \simeq a$ can be mapped to the same propositional variable.

step is thus necessary in order to obtain sets of prime implicates in first-order logic with equality. Note that the experimental comparison in Section 6 does not take the pre- and post-processing steps into account.

The calculus presented in this paper is a form of Constrained Superposition calculus. Other such calculi (see, e.g., Althaus, Kruglov, & Weidenbach, 2009; Bachmair, Ganzinger, & Waldmann, 1994; Baumgartner & Waldmann, 2013) have been defined to handle heterogenous problems, i.e., problems involving a combination of standard first-order logical reasoning with reasoning in specific theories such as arithmetic[16]. In these approaches, constraints are used to isolate the part of the formula belonging to the considered theory from the first-order part which can be handled by the Superposition calculus. External systems can be used afterwards to test the satisfiability of such constraints. Our work departs from these approaches, because constraints are not used to postpone a part of the reasoning, but rather to collect asserted hypotheses. In particular, the underlying theory and semantics of the constraints are identical to those of the clauses, only the "operational meaning" differs.

It is worth comparing the present procedure with our earlier works (Echenim & Peltier, 2016; Echenim, Peltier, & Tourret, 2013, 2014). The procedures devised by Echenim et al. (2013, 2014) only apply to formulæ that are ground and flat, i.e., when the only atoms are equations between constant symbols. The inference rules used for deriving implicates depart from those used in the present approach because there is no distinction between abduced literals and standard ones. Both kinds of literals are handled in a completely uniform way, and the same rules are used both to detect contradictions and to add new hypotheses. On the one hand this may yield more compact representations of the clauses, but on the other hand this results in a less efficient calculus, because most of the restrictions of the Superposition calculus (e.g., ordering constraints) have to be dismissed to ensure deductive completeness. The method described by Echenim and Peltier (2016) uses an explicit representation of abducible literals as constraints, as in the present paper, however such abducible hypotheses are generated in a completely different way. Instead of using specific assertion rules to explicitly add new hypotheses in the derivations, this method works by considering residuals of failed unification attempts as abductive hypotheses. Consider for instance the formula $f(a) \not\simeq f(b)$. This method tries to unify $f(a)$ and $f(b)$ so that the Reflection rule can be applied; it fails (a and b denote distinct constants), but generates in the process a residual equation $a \simeq b$ which cannot be solved. Adding this equation $a \simeq b$ in the constraint allows one to ensure that the inference is applicable, under this additional hypothesis. In contrast, the present approach will first rewrite the constant b into a, adding the equation $a \simeq b$ in the constraint and then apply the Reflection rule on $f(a) \not\simeq f(a)$. The advantage of the former method lies in the fact that it avoids any blind rewriting of the terms (in the above example a could be rewritten to any ground term smaller than a, see the description of the calculus in Section 4 and Remark 4.8 for more details). The use of unification failures allows one to guide the discovery of the additional hypotheses that must be added to generate the empty clause. The advantage of the latter approach is that all the usual restrictions of the Superposition calculus are preserved. In the calculus by Echenim and Peltier (2016) these restrictions must be relaxed to ensure completeness. There is a trade-off between these two

16. Which, usually, cannot be (efficiently) axiomatized.

features. The experimental comparison shows that the latter approach seems to outperform the first one, at least for ground flat clauses (see Echenim et al., 2015; Tourret, 2016).

7.4 Future Work

The most promising direction in which to pursue the work presented in this paper is to implement a version of `cSP` based on a state-of-the-art equational theorem prover such as the `E` theorem prover to benefit from all the clever improvements it contains. The extension of the implementation to the non-ground case also deserves to be considered. Furthermore, our redundancy detection algorithms could be extended to handle variables[17].

Finally, the method could be extended to handle reasoning modulo theories. Two approaches can be considered. The first one consists in adding new axioms or inference rules to handle the theory-specific symbols, as done by, e.g., Waldmann (2001) for totally ordered divisible abelian groups. The other approach consists in handling theory reasoning within the constraints, as done by, e.g., Althaus et al. (2009), Baumgartner and Waldmann (2013), and using an external SMT solver to suggest simplifications, equivalences and abductive hypotheses for the terms of the considered theory.

Acknowledgments

The authors would like to thank the reviewers, for their insightful comments, as well as the creators of the external tools that were used during the experiments, for making them available. Sophie Tourret gratefully acknowledges financial support from the National Institute of Informatics in Tokyo, Japan, at the time the paper was written, and from the University Grenoble Alpes, LIG, in Grenoble, France, at the time the research was conducted.

References

Althaus, E., Kruglov, E., & Weidenbach, C. (2009). Superposition modulo linear arithmetic SUP(LA). In Ghilardi, S., & Sebastiani, R. (Eds.), *Proceedings of the 7th International Symposium on Frontiers of Combining Systems*, Vol. 5749 of *Lecture Notes in Artificial Intelligence*, pp. 84–99. Springer.

Armando, A., Bonacina, M. P., Ranise, S., & Schulz, S. (2009). New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, *10*(1), 1–51.

Baader, F., & Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

Bachmair, L., & Ganzinger, H. (1994). Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, *3*(4), 217–247.

Bachmair, L., Ganzinger, H., & Waldmann, U. (1994). Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, *5*, 193–212.

17. Note however that the entailment relation is undecidable for non-ground clauses, even without equality (Schmidt-Schauss, 1988).

Barrett, C., Fontaine, P., & Tinelli, C. (2015). The SMT-LIB standard: Version 2.5. Tech. rep., Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

Baumgartner, P., & Waldmann, U. (2013). Hierarchic superposition with weak abstraction. In Bonacina, M. P. (Ed.), *Proceedings of the 24th International Conference on Automated Deduction*, Vol. 7898 of *Lecture Notes in Computer Science*, pp. 39–57. Springer.

Bienvenu, M. (2007). Prime implicates and prime implicants in modal logic. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, Vol. 1, pp. 379–384. AAAI Press.

Blackburn, P., Van Benthem, J., & Wolter, F. (2007). *Handbook of Modal Logic*. Studies in logic and practical reasoning. Elsevier.

Bonacina, M. P., & Echenim, M. (2010). Theory decision by decomposition. *Journal of Symbolic Computation*, *45*(2), 229–260.

Cimatti, A., Griggio, A., & Sebastiani, R. (2011). Computing small unsatisfiable cores in Satisfiability Modulo Theories. *Journal of Artificial Intelligence Research*, *40*, 701–728.

Cruanes, S. (2014). Logtk: a logic toolkit for automated reasoning and its implementation. In *4th Workshop on Practical Aspects of Automated Reasoning.*

Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, *17*, 229–264.

De Kleer, J. (1992). An improved incremental algorithm for generating prime implicates. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 780–780. John Wiley & Sons ltd.

De Moura, L. M., & Bjorner, N. (2007). Efficient E-matching for SMT solvers. In Pfenning, F. (Ed.), *Proceedings of the 21st International Conference on Automated Deduction*, Vol. 4603 of *Lecture Notes in Computer Science*, pp. 183–198. Springer.

Dershowitz, N. (1979). Orderings for term-rewriting systems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pp. 123–131, Washington, DC, USA. IEEE Computer Society.

Dershowitz, N., & Manna, Z. (1979). Proving termination with multiset orderings. *Communications of the ACM*, *22*(8), 465–476.

Dillig, I., & Dillig, T. (2013). Explain: a tool for performing abductive inference. In *Proceedings of the 25th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pp. 684–689. Springer.

Dillig, I., Dillig, T., McMillan, K. L., & Aiken, A. (2012). Minimum satisfying assignments for SMT. In Madhusudan, P., & Seshia, S. A. (Eds.), *Computer Aided Verification*, No. 7358 in Lecture Notes in Computer Science, pp. 394–409. Springer.

Echenim, M., & Peltier, N. (2012). A calculus for generating ground explanations. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, Vol. 7364 of *Lecture Notes in Computer Science*, pp. 194–209. Springer.

Echenim, M., Peltier, N., & Tourret, S. (2013). An approach to abductive reasoning in equational logic. In *Proceedings of the 23d International Joint Conference on Artificial Intelligence*, pp. 531–537. AAAI Press.

Echenim, M., & Peltier, N. (2012). An instantiation scheme for Satisfiability Modulo Theories. *Journal of Automated Reasoning*, *48*(3), 293–362.

Echenim, M., & Peltier, N. (2016). A Superposition calculus for abductive reasoning. *Journal of Automated Reasoning*, *57*(2), 97–134.

Echenim, M., Peltier, N., & Tourret, S. (2014). A rewriting strategy to generate prime implicates in equational logic. In *Proceedings of the 7th International Joint Conference on Automated Reasoning*, pp. 137–151. Springer.

Echenim, M., Peltier, N., & Tourret, S. (2015). Quantifier-free equational logic and prime implicate generation. In *Proceedings of the 25th International Conference on Automated Deduction*, pp. 311–325. Springer.

Eiter, T., & Gottlob, G. (1995). The complexity of logic-based abduction. *Journal of the ACM*, *42*(1), 3–42.

Fredkin, E. (1960). Trie memory. *Communications of the ACM*, *3*(9), 490–499.

Ganzinger, H., & Korovin, K. (2003). New directions in instantiation-based theorem proving. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science,(LICS'03)*, pp. 55–64. IEEE Computer Society Press.

Ge, Y., & Moura, L. M. D. (2009). Complete instantiation for quantified formulas in Satisfiability Modulo Theories. In Bouajjani, A., & Maler, O. (Eds.), *Proceedings of the 21st International Conference on Computer Aided Verification*, Vol. 5643 of *Lecture Notes in Computer Science*, pp. 306–320. Springer.

Inoue, K. (2016). Meta-level abduction. *IfCoLog Journal of Logics and their Applications*, *3*(1), 7–35.

Iwanuma, K., Nabeshima, H., & Inoue, K. (2009). Toward an efficient equality computation in connection tableaux: a modification method without symmetry transformation—a preliminary report—. *Proceedings of the international workshop on First-order Theorem Proving*, *556*, 19.

Jackson, P. (1992). Computing prime implicates incrementally. In *Proceedings of the 11th International Conference on Automated Deduction*, pp. 253–267. Springer.

Jouannaud, J., & Kirchner, C. (1991). Solving equations in abstract algebras: a rule based survey of unification. In Lassez, J.-L., & Plotkin, G. (Eds.), *Essays in Honor of Alan Robinson*, pp. 91–99. The MIT-Press.

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

Knill, E., Cox, P. T., & Pietrzykowski, T. (1993). Equality and abductive residua for Horn clauses. *Theoretical Computer Science*, *120*(1), 1–44.

Leitsch, A. (1997). *The resolution calculus*. Texts in Theoretical Computer Science. Springer.

Liberatore, P. (2005). Redundancy in logic I: CNF propositional formulae. *Artificial Intelligence*, *163*(2), 203–232.

Marquis, P. (1991). Extending abduction from propositional to first-order logic. In *Proceedings of the International Workshop on Fundamentals of Artificial Intelligence Research*, pp. 141–155. Springer.

Marquis, P. (2000). Consequence finding algorithms. In *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, pp. 41–145. Springer.

Matusiewicz, A., Murray, N. V., & Rosenthal, E. (2009). Prime implicate tries. In Giese, M., & Waaler, A. (Eds.), *Proceedings of the 18th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pp. 250–264. Springer.

Matusiewicz, A., Murray, N. V., & Rosenthal, E. (2011). Tri-based set operations and selective computation of prime implicates. In Kryszkiewicz, M., Rybinski, H., Skowron, A., & Raś, Z. W. (Eds.), *Proceedings of the 19th International Symposium on Methodologies for Intelligent Systems*, pp. 203–213. Springer.

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

McCune, W. (2005–2010). Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`.

Mishchenko, A. (2001). An introduction to zero-suppressed binary decision diagrams. Tech. rep., In *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*.

Nabeshima, H., Iwanuma, K., Inoue, K., & Ray, O. (2010). SOLAR: an automated deduction system for consequence finding. *AI Communications*, *23*(2), 183–203.

Ngair, T. (1993). A new algorithm for incremental prime implicate generation. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Vol. 1, pp. 46–51. Morgan Kaufmann Publishers Inc.

Nieuwenhuis, R., & Rubio, A. (2001). Paramodulation-based theorem proving. In Robinson, J. A., & Voronkov, A. (Eds.), *Handbook of Automated Reasoning*, pp. 371–443. Elsevier and MIT Press.

Patel-Schneider, P. F., Hayes, P., Horrocks, I., et al. (2004). OWL web ontology language semantics and abstract syntax. *W3C recommendation*, *10*.

Previti, A., Ignatiev, A., Morgado, A., & Marques-Silva, J. (2015). Prime compilation of non-clausal formulae. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pp. 1980–1987. AAAI Press.

Quine, W. (1955). A way to simplify truth functions. *The American Mathematical Monthly*, *62*(9), 627–631.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, *12*, 23–41.

Rougny, A., Yamamoto, Y., Nabeshima, H., Bourgne, G., Poupon, A., Inoue, K., & Froidevaux, C. (2015). Completing signaling networks by abductive reasoning with perturbation experiments. In *Proceedings of the 25th International Conference on Inductive Logic Programming*.

Schmidt-Schauss, M. (1988). Implication of clauses is undecidable. *Theoretical Computer Science*, *59*(3), 287 – 296.

Schulz, S. (2013). System Description: E 1.8. In McMillan, K., Middeldorp, A., & Voronkov, A. (Eds.), *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Stellenbosch*, Vol. 8312 of *Lecture Notes in Computer Science*, pp. 735–743. Springer.

Simon, L., & Del Val, A. (2001). Efficient consequence finding. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pp. 359–370.

Sutcliffe, G. (2009). The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, *43*(4), 337–362.

Tison, P. (1967). Generalization of consensus theory and application to the minimization of boolean functions. *IEEE Transactions on Electronic Computers*, *EC-16*(4), 446–456.

Tourret, S. (2016). *Prime Implicate Generation in Equational Logic*. Ph.D. thesis, Ecole Doctorale MSTII, Grenoble Alpes University.

Voronkov, A. (1995). The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Atomated Reasoning*, *15 (2)*, 237–265.

Waldmann, U. (2001). Superposition and chaining for totally ordered divisible abelian groups. In Goré, R., Leitsch, A., & Nipkow, T. (Eds.), *Proceedings of the 1st International Joint Conference of Automated Reasoning*, Vol. 2083 of *Lecture Notes in Computer Science*, pp. 226–241. Springer.

Weidenbach, C., Afshordel, B., Brahm, U., Cohrs, C., Engel, T., Keen, E., Theobalt, C., & Topic, D. (2001). System description: SPASS version 1.0.0. In *Proceedings of the 16th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pp. 378–382. Springer.