

A Superposition-Based Approach to Abductive Reasoning in Equational Clausal Logic

M. Echenim, N. Peltier, and S. Tournet

Univ. Grenoble Alpes, F-38000 Grenoble, France
CNRS, LIG

Abstract. Abduction can be defined as the process of inferring plausible explanations (or hypotheses) from observed facts (conclusions). This form of reasoning has the potential of playing a central role in system verification, particularly for identifying bugs and providing hints to correct them. We describe an approach to perform abductive reasoning that is based on the superposition calculus. The formulas we consider are sets of first-order clauses with equality, and the explanations that are allowed to be inferred are boolean combinations of equations constructed over a given finite set of ground terms. By duality, abduction can be reduced to a consequence-finding problem. We show how the inference rules of the superposition calculus can be adapted to obtain a calculus that is deductive complete for ground clauses built on the considered sets of ground terms, thus guaranteeing that all required explanations can be generated. This calculus enjoys the same termination properties as the superposition calculus: in particular, it is terminating on ground extensions of decidable theories of interest in software verification.

The number of implicates of a given equational formula is usually huge. We describe techniques for storing sets of abduced clauses efficiently, and show how usual trie-based approaches for representing sets of propositional clauses in a compact way can be adapted and extended in order to denote equational clauses up to equivalence (modulo the axioms of equality). We provide algorithms for performing redundancy pruning in an efficient way on such representations. We also identify hints for improvements and provide lines of on-going and future research.

1 Introduction

The verification of complex systems is often based on proving the validity, or, dually, the satisfiability of a logical formula. The behavior of the system to be verified and its intended properties are both translated into logical formulas ϕ and ψ respectively, and the verification task boils down to proving that $\phi \Rightarrow \psi$ holds, i.e., that the formula $\phi \wedge \neg\psi$ is unsatisfiable. If the formula turns up to be satisfiable, then any model of $\phi \wedge \neg\psi$ can be viewed as a trace that generates an error. Such models help system designers locate the origin of the errors and provide hints to correct them, and most state-of-the-art SMT solvers (see for instance <http://www.smtlib.org/>) feature automated model building tools (see

[7] for more details about general-purpose model building procedures in first-order logic). However, this approach is not always satisfactory. First, there is the risk of an information overkill: indeed, the generated model may be very large and complex, and discovering the origin of the error may require a long and difficult analysis. Second, the model may be too specific, in the sense that it only corresponds to one particular execution of the system and that dismissing this single execution may not be sufficient to fix the system. There are generally many interpretations on different domains that satisfy the formula. In order to understand where the error(s) may come from, it is generally necessary to analyze many of these models and to identify common patterns. This leaves the user with the burden of having to infer the general property that can rule out all the undesired behaviors.

We present what is, to the best of our knowledge, a novel approach to this debugging problem. Rather than studying one or several models of a formula, more valuable information can be extracted from the properties that hold in *all* the models of the formula. Our goal is thus to directly infer the missing axioms, or hypotheses, that can be added in order to ensure the unsatisfiability of the input formula. These axioms can be viewed as sufficient conditions ensuring that the system is valid. Such conditions must be *plausible* and *economical*: for instance, explanations that contradict the axioms of the considered theories are obviously irrelevant. We distinguish a specific set of ground terms on which additional hypotheses are allowed to be made. These terms may be represented by a particular set of constant symbols, called *abducible constants* or simply *abducibles*. The problem boils down to determining what ground clauses containing only abducible constants are logically entailed by the formula under consideration. The negation of any of these clauses can then be viewed as a set of additional hypotheses that make the formula unsatisfiable. Indeed, by duality, computing implicants (or explanations) of a formula ϕ is equivalent to computing implicates (i.e., logical consequences) of $\neg\phi$.

The generation of implicants (or, by duality, of implicates) of logical formulas has many applications in system verification and artificial intelligence, and this problem has been thoroughly investigated in the context of *propositional* logic. The earlier approaches use refinements of the resolution method [32, 20, 9, 31], while more recent and more efficient proposals use decomposition-based procedures [19, 8, 18, 25, 26]. These methods mainly focus on the efficient representation of information, and develop compact ways of storing and manipulating huge sets of implicates. In contrast, the approaches handling abductive reasoning in first-order or equational logic are very scarce [24, 21, 27, 11, 29]. When dealing with equational clause sets, the addition of equality axioms leads to inefficiency and divergence in almost all but trivial cases (note that the transitivity axiom alone already admits infinitely many implicates!).

In order to compute equational implicates in an efficient way, we devise a variant of the superposition calculus [4, 30] that is deductive-complete for the considered set of abducible constants, i.e., that can generate all the clauses built on abducible constants that are logical consequences of the input clause set, up

to redundancy. Our procedure is defined by enriching the standard calculus with some new mechanisms allowing the assertion of relevant hypotheses during the proof search. These additional hypotheses are stored as constraints associated with the clauses and are propagated along the derivations. If the empty clause can be generated under a conjunction of hypotheses $\mathcal{X}$, then the conjunction of the original formula and $\mathcal{X}$ is unsatisfiable. An essential feature of this approach is that the conditions are not asserted arbitrarily or eagerly, using a generate-and-test approach (which would be inefficient): instead they are *discovered* on a need basis.

Due to a lack of space, proofs are omitted. Some of the proofs can be found in [12, 16, 14].

2 Preliminaries

The set of *terms* T is built as usual on a set of *function symbols* $\mathcal{F}$ and a set of *variables* $\mathcal{V}$. Every symbol $f \in \mathcal{F}$ is mapped to a unique *arity* $ar(f) \in \mathbb{N}$. An element of $\mathcal{F}$ of arity 0 is a *constant*. An *atom* (or *equation*) is an unordered pair of terms, written $t \simeq s$, where t and s are terms. A *literal* is either an atom or the negation of an atom (i.e., a *disequation*), written $t \not\simeq s$. For every literal l , we denote by l^c the complementary literal to l , which is defined as follows: $(t \simeq s)^c \stackrel{\text{def}}{=} t \not\simeq s$ and $(t \not\simeq s)^c \stackrel{\text{def}}{=} t \simeq s$. As usual, a non-equational atom $p(\mathbf{t})$ is encoded as an equation $p(\mathbf{t}) \simeq \mathbf{true}$ (where $\mathbf{true}$ is a special constant symbol). For readability, such an equation is sometimes written $p(\mathbf{t})$, and $p(\mathbf{t}) \not\simeq \mathbf{true}$ is written $\neg p(\mathbf{t})$. A *clause* is a finite multiset of literals, sometimes written as a disjunction. The empty clause is denoted by $\square$. For every clause $C = \{l_1, \dots, l_n\}$, C^c denotes the set of unit clauses $\{\{l_i^c\} \mid i \in [1, n]\}$ and for every set of unit clauses $S = \{\{l_i\} \mid i \in [1, n]\}$, S^c denotes the clause $\{l_1^c, \dots, l_n^c\}$.

Let $\mathcal{F}_i \subseteq \mathcal{F}$ be a set of *interpreted function symbols*, and let $\mathcal{A}$ be a set of constants, called the *abducible constants*. The set $\mathcal{F}_i$ contains symbols whose interpretation is fixed according to the considered theory (e.g. $\mathcal{F}_i$ may contain symbols 0, 1, + ... interpreted as natural numbers and operations on them), whereas the symbols in $\mathcal{F} \setminus \mathcal{F}_i$ are interpreted arbitrarily. The set $\mathcal{A}$ is fixed by the user and contains all constants on which the abducible formulas can be constructed. Such constants are used as names for ground terms, and equations of the form $c \simeq t$ can be added to encode the fact that an abducible constant c denotes the ground term t .

The set of variables occurring in an expression (term, atom, literal, clause) E is denoted by $\text{var}(E)$. If $\text{var}(E) = \emptyset$ then E is *ground*. Let $\mathcal{V}_a, \mathcal{V}_i \subseteq \mathcal{V}$ be special sets of variables which are allowed to be instantiated only by constants in $\mathcal{A}$ and by terms built on $\mathcal{F}_i$, respectively. A *substitution* σ is a function mapping every variable to a term, such that if $x \in \mathcal{V}_a$ then $\sigma(x) \in \mathcal{A} \cup \mathcal{V}_a$ and if $x \in \mathcal{V}_i$ then $\sigma(x)$ is a term built on $\mathcal{F}_i \cup \mathcal{V}_i$. For every term t and for every substitution σ , we denote by $t\sigma$ the term obtained from t by replacing every variable x by its image w.r.t. σ . The *domain* of a substitution is the set of variables x such that $x\sigma \neq x$. A substitution σ is *ground* if for every x in the domain of σ , $x\sigma$ is ground.

Throughout the paper, we assume that $\prec$ denotes some fixed reduction ordering on terms such that $\mathbf{true} \prec t$, for all terms $t \neq \mathbf{true}$. The ordering $\prec$ is extended to atoms, literals and clauses as usual (see, e.g., [4]). We assume that $f(\mathbf{t}) \succ a$, for all $a \in \mathcal{A}$ and all functions and constants $f \notin \mathcal{A} \cup \mathcal{F}_i$, and, similarly, that $f(\mathbf{s}) \succ t$, for all terms t built on $\mathcal{F}_i$ and $\mathcal{V}_i$ and for all functions and constants $f \notin \mathcal{F}_i$. This entails that we can also assume that $f(\mathbf{s}) \succ x$, for every $x \in \mathcal{V}_a$ and $f \notin \mathcal{A} \cup \mathcal{F}_i$ or $x \in \mathcal{V}_i$ and $f \notin \mathcal{F}_i$.

A *position* is a finite sequence of natural numbers. A position p *occurs* in a term t if either $p = \varepsilon$ or if $t = f(t_1, \dots, t_n)$, $p = i.q$ with $i \in [1, n]$ and q is a position in t_i . If p is a position in t , the terms $t|_p$ and $t[s]_p$ are defined as follows: $t|_\varepsilon \stackrel{\text{def}}{=} t$, $t[s]_\varepsilon \stackrel{\text{def}}{=} s$, $f(t_1, \dots, t_n)|_{i.q} \stackrel{\text{def}}{=} (t_i)|_q$ and $f(t_1, \dots, t_n)[s]_{i.q} \stackrel{\text{def}}{=} f(t_1, \dots, t_{i-1}, t_i[s]_q, t_{i+1}, \dots, t_n)$.

Given a set of constants E , a literal $t \bowtie s$ (with $\bowtie \in \{\simeq, \not\simeq\}$) is *E-flat* if $t, s \in \mathcal{V} \cup E$. A clause is *E-flat* if all its literals are *E-flat*. The set of *E-flat* clauses is denoted by $\mathcal{C}_{flat}(E)$. An expression is *flat* if it is *E-flat* for some set of constants E .

An *interpretation* is a congruence relation on ground terms, where the interpretation of the symbols in $\mathcal{F}_i$ is fixed according to the considered theory (for instance $0 + 1 \simeq 1$ is to be interpreted as true if $0, 1$ and $+$ denote the usual interpreted symbols on natural numbers, and $n \simeq \underbrace{1 + \dots + 1}_{k \text{ times}}$ is to be interpreted

as true for some $k \in \mathbb{N}$ if n is intended to denote a non-fixed natural number). An interpretation I *validates* a clause C if for all ground substitutions σ of domain $\text{var}(C)$ there exists $l \in C$ such that either $l = (t \simeq s)$ and $(l, s)\sigma \in I$, or $l = (t \not\simeq s)$ and $(l, s)\sigma \notin I$.

Note that $\mathcal{F}_i$ possibly contains interpreted functions whose value is not completely specified (for instance constants denoting arbitrary natural numbers), in which case the unsatisfiability problem becomes non-semi-decidable. The interpretation of the symbols in $\mathcal{F} \setminus \mathcal{F}_i$ is fully arbitrary (as in first-order logic). In the following, completeness results will be given only for the special case in which $\mathcal{F}_i = \emptyset$.

3 A Constrained Superposition Calculus

In this section we define an extension of the standard superposition calculus [4, 30] with which it is possible to generate all $\mathcal{A}$ -flat implicates of a considered clause set. The results in this section are extensions of [13] (see Section 3.2 for a more detailed comparison). The calculus handles constrained clauses, or *c-clauses*, the constraint part of a c-clause being a set containing all the equations and disequations needed to derive the corresponding non-constraint part from the original clause set. The calculus uses the usual rules of the superposition calculus [4]; furthermore, an additional inference rule, called the *A-Assertion* rule, is introduced in order to add disequations to the constraints.

Definition 1. A c-clause is a pair $[C \mid \mathcal{X}]$ where C is a clause containing no symbol in $\mathcal{A} \cup \mathcal{F}_i$ and $\mathcal{X}$ is a set of literals of the form $u \bowtie v$ where both u and

v are terms built on $\mathcal{A} \cup \mathcal{F}_i$ and $\mathcal{V}_a \cup \mathcal{V}_i$. If $\mathcal{X} = \emptyset$, then we may write C instead of $[C \mid \emptyset]$.

A clause containing symbols in $\mathcal{A} \cup \mathcal{F}_i$ can be transformed into an equivalent c-clause by *abstracting away* such symbols and replacing them by variables in $\mathcal{V}_a$ or $\mathcal{V}_i$. For instance, the clause $p(a, 1)$ where $a \in \mathcal{A}$ and $1 \in \mathcal{F}_i$ is written $[p(x, y) \mid \{x \simeq a, y \simeq 1\}]$, with $x \in \mathcal{V}_a$, $y \in \mathcal{V}_i$.

Let sel be a *selection function*, mapping every clause C to a set of literals in C such that $sel(C)$ either contains a negative literal or contains all literals that are $\preceq$ -maximal in C . We assume that sel is stable under substitutions, i.e., that for every clause C , for every literal $l \in C$ and for every substitution η , if $l\eta \in sel(C\eta)$, then $l \in sel(C)$.

3.1 Inference Rules

The calculus $\mathcal{SA}_{sel}^\prec$ is defined by the rules below. The standard superposition calculus (denoted by $\mathcal{SP}_{sel}^\prec$) coincides with $\mathcal{SA}_{sel}^\prec$ if $\mathcal{A} = \mathcal{F}_i = \emptyset$.

$\mathcal{A}$ -Superposition

$$\frac{[C \vee t \bowtie s \mid \mathcal{X}], [D \vee u \simeq v \mid \mathcal{Y}]}{[C \vee D \vee t[v]_p \bowtie s \mid \mathcal{X} \cup \mathcal{Y}]\sigma}$$

If $\bowtie \in \{\simeq, \neq\}$, σ is a most general unifier of u and $t|_p$, $v\sigma \not\prec u\sigma$, $s\sigma \not\prec t\sigma$, $(t \bowtie s)\sigma \in sel((C \vee t \bowtie s)\sigma)$, $(u \simeq v)\sigma \in sel((D \vee u \simeq v)\sigma)$ and if $t|_p$ is a variable then $t|_p \in \text{var}(\mathcal{X}) \cap \mathcal{V}_a$.

The main difference with the usual superposition rule is that superposition into a variable is permitted, provided the considered variable occurs in the constraint part of the clause. The reason is that these symbols do not actually represent variables in the usual sense, but rather placeholders for (unknown) constants.

$\mathcal{A}$ -Reflection

$$\frac{[C \vee t \neq s \mid \mathcal{X}]}{[C \mid \mathcal{X}]\sigma}$$

If σ is a most general unifier of t and s and $(t \neq s)\sigma \in sel((C \vee t \neq s)\sigma)$.

Equational $\mathcal{A}$ -Factorization

$$\frac{[C \vee t \simeq s \vee u \simeq v \mid \mathcal{X}]}{[C \vee s \neq v \vee t \simeq s \mid \mathcal{X}]\sigma}$$

If σ is a most general unifier of t and u , $s\sigma \not\prec t\sigma$, $v\sigma \not\prec u\sigma$ and $(t \simeq s)\sigma \in sel((C \vee t \simeq s \vee u \simeq v)\sigma)$.

$\mathcal{A}$ -Assertion

$$\frac{[x \simeq y \vee C \mid \mathcal{X}]}{[C \mid \mathcal{X} \cup \{x \not\simeq y\}]}$$

If $x, y \in \mathcal{V}_a \cup \mathcal{V}_i$, $x \simeq y \in \text{sel}(x \simeq y \vee C)$ and $\mathcal{A} \cup \mathcal{F}_i \neq \emptyset$.

If $\mathcal{F}_i = \emptyset$, then the calculus is essentially equivalent to the one defined in the research report [14] although the presentation is slightly different (the calculus in [14] does not abstract away the constant symbols in $\mathcal{A}$, instead it performs unification modulo equations between such constants – in other words the replacement of constants in $\mathcal{A}$ by variables is directly handled by the unification algorithm).

Redundancy We now adapt the standard redundancy criterion to c-clauses.

Definition 2. A c-clause $[C \mid \mathcal{X}]$ is $\mathcal{A}$ -redundant in a set of c-clauses S if for every ground substitution θ of the variables in $[C \mid \mathcal{X}]$ such that $\mathcal{X}\theta$ is satisfiable, there exist c-clauses $[D_i \mid \mathcal{Y}_i]$ and substitutions σ_i ($1 \leq i \leq n$), such that:

- $\mathcal{Y}_i\sigma_i \models \mathcal{X}\theta$ for all $i = 1, \dots, n$;
- $D_1\sigma_1, \dots, D_n\sigma_n \models C\theta$;
- $D_1\sigma_1, \dots, D_n\sigma_n \preceq C\theta$.

Definition 3. A set S is $\mathcal{SA}_{\text{sel}}^{\prec}$ -saturated if every c-clause that can be derived from c-clauses in S by a rule in $\mathcal{SA}_{\text{sel}}^{\prec}$ is redundant in S .

3.2 Soundness and Deductive Completeness

The interpretation of a c-clause is defined from its set of ground instances, viewing constraints as logical implications:

Definition 4. An interpretation I validates a c-clause $[C \mid \mathcal{X}]$ iff for every ground substitution σ of domain $\text{var}(C) \cup \text{var}(\mathcal{X})$, either $I \not\models \mathcal{X}\sigma$ or $I \models C\sigma$.

The following theorem states the soundness of $\mathcal{SA}_{\text{sel}}^{\prec}$.

Theorem 1. Let S be a set of c-clauses. If C is deducible from S by one of the rules of $\mathcal{SA}_{\text{sel}}^{\prec}$ then $S \models C$.

In the remainder of this section, we assume that $\mathcal{F}_i$ is empty, i.e., that the considered clause set contains no interpreted symbol. Note that the logic is not semi-decidable in general in the presence of interpreted symbols, thus no procedure can be refutationally complete if $\mathcal{F}_i \neq \emptyset$.

Definition 5. Let S be a set of c-clauses. A clause C is an $\mathcal{A}$ -implicate of S if it satisfies the following conditions.

- C is $\mathcal{A}$ -flat and ground.
- C is not a tautology.

- $S \models C$.

The clause C is a prime $\mathcal{A}$ -implicate of S if, moreover, $C \models D$ holds for every $\mathcal{A}$ -implicate D of S such that $D \models C$. We denote by $I_{\mathcal{A}}(S)$ the set of $\mathcal{A}$ -implicates of S .

Definition 6. We denote by $\mathcal{C}_{\mathcal{A}}(S)$ the set of clauses of the form $(\mathcal{X}\sigma)^c$, where $[\Box | \mathcal{X}] \in S$ and $\mathcal{X}\sigma$ is satisfiable. We write $S \sqsubseteq S'$ if for every clause $C' \in S'$, there exists $C \in S$ such that $C \models C'$.

The main result we have is that $\mathcal{C}_{\mathcal{A}}(S) \sqsubseteq I_{\mathcal{A}}(S)$ holds for all saturated sets S . However, in practice it is important not only to be able to generate the sets of *all* prime $\mathcal{A}$ -implicates but also to be able to generate only some *specific* $\mathcal{A}$ -implicates, satisfying some additional property (e.g., to generate unit or positive implicates, etc.). To compute such sets of implicates efficiently, it is essential to block inferences generating candidates not satisfying the desired property. We thus introduce the following definition.

Definition 7. A set of clauses $\mathfrak{P}$ is closed under subsumption if for every $C \in \mathfrak{P}$ and for every clause D such that D subsumes C , we have $D \in \mathfrak{P}$. A c -clause $[C | \mathcal{X}]$ is $\mathfrak{P}$ -compatible if $\mathcal{X}^c \in \mathfrak{P}$. $\mathcal{SA}_{sel}^{\prec}(\mathfrak{P})$ denotes the calculus $\mathcal{SA}_{sel}^{\prec}$ in which all inferences that generate non- $\mathfrak{P}$ -compatible c -clause are blocked.

Examples of classes of clauses that are closed under subsumption include the following sets that are of some practical interest:

- The set of clauses C such that there exists a substitution σ such that $C\sigma$ is equivalent to a clause of length at most k .
- The set of positive (resp. negative) clauses.
- The set of implicants of some formula ϕ .

Note also that the class of clause sets that are closed under subsumption is closed under union and intersection, which entails that these criteria can be combined easily.

Theorem 2. Let S_{init} be a set of standard clauses and let S be a set of c -clauses obtained from S_{init} by $\mathcal{SA}_{sel}^{\prec}(\mathfrak{P})$ -saturation. If $\mathfrak{P}$ is closed under subsumption then $\mathcal{C}_{\mathcal{A}}(S) \sqsubseteq I_{\mathcal{A}}(S) \cap \mathfrak{P}$.

In particular, if $\mathfrak{P}$ is the set of all clauses constructed over $\mathcal{A}$, then $\mathcal{SA}_{sel}^{\prec}$ -saturation permits to obtain the set of all prime $\mathcal{A}$ -implicates of a given set of clauses

A concise representation of $\mathcal{A}$ -implicates. It is interesting to investigate what happens if one prevents inferences on $\mathcal{A}$ -literals. The obtained calculus then essentially simulates the one introduced in [13]. It is not complete since it does not generate all $\mathcal{A}$ -implicates in general, but it is complete in a restricted sense: every $\mathcal{A}$ -implicate is a logical consequence of the set of $\mathcal{A}$ -flat clauses generated by the calculus (note that [13] established this result only in the particular case of

variable-inactive clauses [2]). More precisely, we denote by $\mathcal{SAR}_{sel}^{\prec}$ the calculus $\mathcal{SA}_{sel}^{\prec}$ in which no inference upon $\mathcal{A}$ -literals is allowed, except for the $\mathcal{A}$ -assertion and $\mathcal{A}$ -reflection rules.

Theorem 3. *For all $\mathcal{SAR}_{sel}^{\prec}$ -saturated sets of c-clauses S , we have $\mathcal{C}_{\mathcal{A}}(S) \models I_{\mathcal{A}}(S)$.*

The difference between the calculi $\mathcal{SA}_{sel}^{\prec}$ and $\mathcal{SAR}_{sel}^{\prec}$ can be summarized as follows.

- The calculus $\mathcal{SA}_{sel}^{\prec}$ explicitly generates all prime implicants in $I_{\mathcal{A}}(S)$, whereas $\mathcal{SAR}_{sel}^{\prec}$ only generates a finite representation of these implicants, in the form of an $\mathcal{A}$ -flat implicant S' of $I_{\mathcal{A}}(S)$. The formula S' can still contain redundancies and some additional post-processing step is required to generate explicitly the prime implicants of S' if needed. Any algorithm for generating prime implicants of propositional clause sets can be used for this purpose, since flat ground equational clause sets can be reduced into equivalent sets of propositional clauses by adding equality axioms. From a practical point of view, the set $I_{\mathcal{A}}(S)$ can be very large, thus S' can also be viewed as a concise and suitable representation of this set.
- The calculus $\mathcal{SAR}_{sel}^{\prec}$ restricts inferences on $\mathcal{A}$ -flat literals to those that actually delete such literals, possibly by transferring them to the constraint part of the clauses (the $\mathcal{A}$ -Assertion and $\mathcal{A}$ -Reflection rules). From a practical point of view, this entails that these literals do not need to be considered anymore in the clausal part of the c-clause: they can be transferred *systematically* in the constraints. This can reduce the number of generated clauses by an exponential factor, since a given $\mathcal{A}$ -flat clause $l_1 \vee \dots \vee l_n$ can be in principle represented by 2^n distinct c-clauses depending on whether l_i is stored to the clausal or constraint part of the c-clause (for instance $a \simeq b$ can be represented as $[a \simeq b \mid \emptyset]$ or $[\Box \mid a \not\simeq b]$). Furthermore, the number of applicable inferences is also drastically reduced, since the rules usually apply in many different ways on (selected) $\mathcal{A}$ -literals.

It is possible to combine the two calculi $\mathcal{SA}_{sel}^{\prec}$ and $\mathcal{SAR}_{sel}^{\prec}$. This can be done as follows.

- Starting from a set of clauses S , $\mathcal{SAR}_{sel}^{\prec}$ is applied first until saturation, yielding a new set S' . By Theorem 3 we have $\mathcal{C}_{\mathcal{A}}(S') \equiv I_{\mathcal{A}}(S)$.
- Then $\mathcal{SA}_{sel}^{\prec}(\mathfrak{P})$ is applied on $\mathcal{C}_{\mathcal{A}}(S')$ until saturation yielding a set S'' , where $\mathfrak{P}$ denotes the set of clauses that logically entail at least one clause in $\mathcal{C}_{\mathcal{A}}(S')$. It is clear that this set of clauses is closed under subsumption, hence by Theorem 2, we eventually obtain a set of clauses $\mathcal{C}_{\mathcal{A}}(S'') \sqsubseteq I_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}}(S')) \cap \mathfrak{P}$. But $I_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}}(S')) \cap \mathfrak{P} \sqsubseteq \mathcal{C}_{\mathcal{A}}(S')$, hence $\mathcal{C}_{\mathcal{A}}(S'') \sqsubseteq \mathcal{C}_{\mathcal{A}}(S')$, and $\mathcal{C}_{\mathcal{A}}(S'') \equiv I_{\mathcal{A}}(S)$. The set of clauses $\mathcal{C}_{\mathcal{A}}(S'')$ can therefore be considered as a concise representation of $I_{\mathcal{A}}(S)$. This approach is appealing since $\mathcal{C}_{\mathcal{A}}(S'')$ is in generally much smaller than $I_{\mathcal{A}}(S)$, and contrary to $\mathcal{C}_{\mathcal{A}}(S')$, this set is free of redundancies.

Another straightforward method to eliminate redundant literals from the clauses in $\mathcal{C}_{\mathcal{A}}(S')$ without having to explicitly compute the set $I_{\mathcal{A}}(S')$ is to test, for every

clause $l \vee C \in \mathcal{C}_{\mathcal{A}}(S')$, whether the relation $\mathcal{C}_{\mathcal{A}}(S') \models C$, holds, if which case the literal l can be safely removed. The test can be performed by using any decision procedure for ground equational logic (see for instance [28]). See [10] for a similar approach.

Another calculus was proposed in [15] to generate implicates of ground flat clauses. This calculus is very similar to the restriction of $\mathcal{SA}_{sel}^<$ to abstracted flat clauses, except that constraints are attached to the clauses as additional literals (e.g., a c-clause $[x \simeq y | \{x \not\simeq a, y \not\simeq b, c \simeq d\}]$ is represented as the clause $a \simeq b \vee c \not\simeq d$). As a consequence, several additional inferences are possible, because superposition inferences can be performed on literals in the constraints, but the total number of clause representatives is reduced as explained above. Furthermore, the redundancy criterion used in [15] is more liberal than the one considered in the present paper: in order to keep the clause sets as compact as possible, all clauses that are logically entailed by an existing clause are considered as redundant¹. An important consequence is that a new inference rule had to be devised to perform superposition inferences simultaneously on several negative literals, so that completeness can be enforced despite this weaker redundancy criterion.

4 Termination Results

In this section we relate the termination behavior of $\mathcal{SA}_{sel}^<$ to that of the usual superposition calculus in the case where $\mathcal{F}_i = \emptyset$. We first introduce restricted ordering and redundancy criteria.

Definition 8. *For all expressions (terms, atoms, literals or clauses) t and s , we write $t \succ_{\mathcal{A}} s$ if $t' \succ s'$ holds for all expressions t', s' that are identical to t and s respectively, up to a renaming of constants in $\mathcal{A}$. We denote by $sel_{\mathcal{A}}$ the selection function defined as follows: for every clause $l \vee C$, $l \in sel_{\mathcal{A}}(l \vee C)$ if there exists l', C' such that $l' \in sel(l' \vee C')$ and l', C' are identical to l and C respectively, up to a renaming of constant symbols in $\mathcal{A}$.*

We show that most termination results for the calculus $\mathcal{SP}_{sel_{\mathcal{A}}}^<^{\mathcal{A}}$ also apply to $\mathcal{SA}_{sel}^<$. To this purpose, we consider a restricted form of redundancy testing.

Definition 9. *An ordinary clause C is strongly redundant in a set of ordinary clauses S iff for every clause C' that is identical to C up to a renaming of constants in $\mathcal{A}$, C' is $\mathcal{A}$ -redundant in S .*

We denote by $\mathcal{E}_{\mathcal{A}}$ the set of unit clauses $a \simeq b$ or $a \not\simeq b$, with $a, b \in \mathcal{A}$. For any set of clauses S , let S^* be the set of clauses inductively defined as follows.

- $S \subseteq S^*$.

¹ The redundancy criterion used in [15] is actually not exactly logical entailment, but a slight restriction of it, so that the factors of a clause are not redundant w.r.t. their premise.

- If C is not strongly redundant in S and is deducible from $S^* \cup \mathcal{E}_{\mathcal{A}}$ by applying the rules in $\mathcal{SP}_{sel_{\mathcal{A}}}^{\prec_{\mathcal{A}}}$ (in one step) then $C \in S^*$.

Theorem 4. *Let S be a set of clauses. If S^* is finite then $\mathcal{SA}_{sel}^{\prec}$ terminates on S (up to redundancy).*

In order to prove that $\mathcal{SA}_{sel}^{\prec}$ terminates on some class of clause sets $\mathfrak{S}$, it suffices to prove that S^* is finite, for every $S \in \mathfrak{S}$. The calculus $\mathcal{SP}_{sel}^{\prec_{\mathcal{A}}}$ is slightly less restrictive than the usual superposition calculus $\mathcal{SP}_{sel}$, since $\prec_{\mathcal{A}}$ is a stronger relation than $\prec$ and $sel_{\mathcal{A}}(C) \supseteq sel(C)$. However, most of the usual termination results for the superposition calculus still hold for $\mathcal{SP}_{sel}^{\prec_{\mathcal{A}}}$, because they are closed under the addition of equalities between constants and do not depend on the order of constant symbols. Similarly, redundancy testing is usually restricted to subsumption and tautology detection. In particular, all the termination results in [3] are preserved (it is easy to check that S^* is finite for the considered sets of axioms).

An interesting continuation of the present work would be to devise formal (automated) proofs of the termination of $\mathcal{SA}_{sel}^{\prec}$ on the usual theories of interest in program verification (enriched by arbitrary ground clauses). This could be done by using existing schematic calculi [22, 23] to compute a symbolic representation of the set of c-clauses S^* .

5 Representation of Ground Flat Clauses Modulo Equality

The number of ground flat implicates of a given formula is often huge, thus it is essential in practice to provide compact representations for such sets, as well as efficient algorithms for detecting and discarding redundant clauses. In propositional logic, detecting redundant clauses is an easy task, because, for all clauses C, D , we have $D \models C$ iff either C is a tautology or $D \subseteq C$. Thus a non-tautological clause C is redundant in a clause set S iff either C contains two complementary literals or if there exists a clause $D \in S$ such that $D \subseteq C$. The clause set S can thus be conveniently represented as a trie (a tree-based data-structure commonly used to represent sets of strings, see for instance [17]), and inclusion can be tested efficiently using standard algorithms (the literals are totally ordered and sorted to handle commutativity of $\vee$). However, in equational logic, the above equivalence does not hold: for example we have $a \simeq c \models a \not\simeq b \vee b \simeq c$ and $a \simeq c \not\models a \not\simeq b \vee b \simeq c$. Representing clause sets as tries would therefore yield many undesired redundancies: for instance the clauses $a \not\simeq b \vee b \simeq c$ and $a \not\simeq b \vee a \simeq c$ would be both stored, although they are equivalent. In this section (adapted from [15]), we provide a new redundancy criterion that generalizes subsumption, together with a new technique for representing clauses, that takes into account the special properties of the equality predicate. Unless stated otherwise, we assume that all the clauses considered in this section are $\mathcal{A}$ -flat and ground.

5.1 Testing Logical Entailment

In what follows, we introduce the notion of C -representativity, which is used to define the normalized form of a clause in such a way that equivalent clauses have the same normalized form, and to define a syntactic test for logical entailment. Let C be a clause.

Definition 10. *The C -representative of a constant a is the constant $a_{|C} \stackrel{\text{def}}{=} \min_{\prec} \{b \in \mathcal{C} \mid \neg C \models b \simeq a\}$ (it is clear that all constants have a representative, since $a \not\simeq a \models C$). This notion extends to literals and clauses as follows: $(a \bowtie b)_{|C} \stackrel{\text{def}}{=} a_{|C} \bowtie b_{|C}$ and $D_{|C} \stackrel{\text{def}}{=} \{l_{|C} \mid l \in D\}$. The expression $E_{|C}$ is called the projection of E on C . We write $E \equiv_C E'$ if $E_{|C} = E'_{|C}$.*

By definition, $\equiv_C$ is an equivalence relation and the following equivalences hold:

$$(a \equiv_C b) \Leftrightarrow (a \not\simeq b \models C) \Leftrightarrow (\neg C \models a \simeq b)$$

The next proposition introduces a notion of normal form for equational clauses. Intuitively, all constants a occurring in a clause C are replaced by their representatives $a_{|C}$, and all inequations $a \simeq a_{|C}$ where $a \neq a_{|C}$ are appended to the clause. This normal form will permit to test efficiently whether a clause is tautological and whether two clauses are equivalent.

Proposition 1. *Every clause C is equivalent to the clause:*

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

Furthermore, C is a tautology iff $C_{\downarrow}$ contains a literal $a \simeq a$. A non-tautological clause C is in normal form if $C = C_{\downarrow}$ and if, moreover, all literals occur at most once in C .

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

Definition 11. *Let C, D be two clauses. The clause D eq-subsumes C (written $D \leq_{eq} C$) iff the two following conditions hold.*

- $\equiv_D \subseteq \equiv_C$ (i.e. every negative literal in $D_{|C}$ is a contradiction).
- For every positive literal $l \in D$, there exists a literal $l' \in C$ such that $l \equiv_C l'$.

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

Intuitively, eq-subsumption consists in verifying that $\neg C \models \neg D$. This is done by first checking that all equations in $\neg D$ are logical consequences of those in $\neg C$, which can be easily done by checking that the relation $\equiv_D \subseteq \equiv_C$ holds. Now, consider a negative literal l in $\neg D$. The literal l can only be entailed by $\neg C$ if $\neg C$ contains a literal l' which can be reduced to l by the relation $\equiv_C$.

Theorem 5. *Let C and D be two clauses. If C is not a tautology then $D \models C$ iff $D \leq_{eq} C$.*

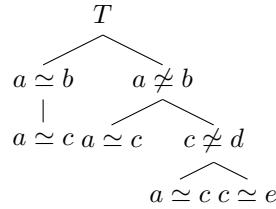
5.2 Clausal Trees

We define data-structures for storing and retrieving generated clauses, in such a way that the redundancy criterion introduced in Section 5.1 can be tested efficiently using Theorem 5. We use for this purpose a tree data-structure, called a *clausal tree*, specifically tailored to store sets of literals while taking into account the specific properties of the equality predicate (among these properties only transitivity is important, since reflexivity and commutativity are easy to handle by using normalization w.r.t. some ordering among constants). As in tries, the edges of the tree are labeled by literals and the leaves are either $\square$ (representing the empty clause) or $\emptyset$ (the failure node, representing an empty set of clauses). Each branch leading to a leaf $\square$ represents a clause defined as the disjunction of the literals labeling the edges in the branch. Failure nodes are useful mainly to represent empty sets – in fact they can always be eliminated by straightforward simplification rules, except if the root itself is labeled by $\emptyset$.

Definition 12. *A clausal tree is inductively defined as either $\square$, or a set of pairs of the form (l, T') where l is a literal and T' a clausal tree. The set of clauses represented by a clausal tree T is denoted by $S_c(T)$ and defined inductively as follows:*

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

Example 1. The structure T below is a clausal tree. There is no failure node, and for readability the labels are associated with the nodes rather than with the edges leading to them.



The represented clauses $S_c(T)$ are:

$$\begin{aligned} a &\simeq b \vee a \simeq c \\ a &\not\simeq b \vee a \simeq c \\ a &\not\simeq b \vee c \not\simeq d \vee a \simeq c \\ a &\not\simeq b \vee c \not\simeq d \vee c \simeq e \end{aligned}$$

We impose additional conditions on the clausal tree, in order to ensure that the represented clauses are in normal form and that sharing is maximal (for instance to ensure that there are no two edges starting from the same node and labeled by the same literal). Furthermore, the literals occurring along a given branch are ordered using the usual multiset extension of $\prec$, with the additional constraint that negative literals always occur before positive ones. The intuition behind this definition is that the negative literals occurring in a clause C specify in some sense the “value” of the constant symbols in C (i.e. their C -representative). Since the representatives of the positive literals depend on these values, it is necessary to ensure that every negative literal is known before positive ones can be stored. More formally, we define an ordering $\triangleleft$ on literals as follows.

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

Definition 13. A clausal tree T is normal if for any pair (l, T') in T , the following conditions hold.

- $T' \neq \emptyset$.
- There is no $T'' \neq T'$ such that $(l, T'') \in T$.
- The literal l is not of the form $a \simeq a$ or $a \not\simeq a$.
- All literals occurring in T' are strictly greater than l w.r.t. $\triangleleft$.
- If $l = a \not\simeq b$ with $a \prec b$ then b does not occur in T' .
- The clausal tree T' is a normal clausal tree.

It is easy to verify that if T is a normal clausal tree then all the clauses in $S_c(T)$ are in normal form.

Normal clausal trees are used to store the set of $\mathcal{A}$ -implicates generated so far, and this set has to be updated every time a more general $\mathcal{A}$ -implicate is generated: all the stored $\mathcal{A}$ -implicates that it entails must be deleted. We now introduce two algorithms for testing whether a newly generated $\mathcal{A}$ -implicate is entailed by one already stored in a clausal tree, and for deleting from a clausal tree all $\mathcal{A}$ -implicates logically entailed by the newly generated one. The first algorithm (ISENTAILED) is invoked on a clause C and a tree T , and returns *true* if and only if there exists a clause D in $S_c(T)$ such that D eq-subsumes C . To test this entailment, the algorithm performs a depth-first traversal of T and attempts to project every encountered literal on C (see Definition 10). If a literal cannot be projected, the exploration of the subtree associated to this literal is useless, so the algorithm switches to the following literal. As soon as a clause entailing C is

found, the traversal halts and *true* is returned. For the sake of readability we use the following notations. For any expression E , $E[a := b]$ denotes the expression obtained from E by replacing all occurrences of a by b . For any clausal tree T and literal l , we denote by $l.T$ the clausal tree $l.T \stackrel{\text{def}}{=} \{(l, T)\}$.

Theorem 6. *The procedure `ISENTAILED` terminates in $\mathcal{O}(\text{size}(S_c(T)) + |C| \times |S_c(T)|)$. Moreover, `ISENTAILED`(C, T) is true iff $S_c(T)$ contains a clause D such that $D \leq_{eq} C$.*

Algorithm 1 `ISENTAILED`(C, T)

```

if  $T = \square$  then
  return true
end if
if  $C = \square$  then
  return false
end if
 $l_1 \leftarrow \min_{\triangleleft} \{l \in C\}$ 
for all  $(l, T') \in T$  such that  $l \geq l_1$  do
  if  $l_1 = a \not\preceq b$ , with  $a \succ b$  then
    if  $l = l_1$  then
      if ISENTAILED( $C \setminus \{l_1\}, T'$ ) then
        return true
      end if
    else if  $\neg(l = a \not\preceq c)$ , with  $a \succ c$  then
      if ISENTAILED( $C \setminus \{l_1\}, (l.T')[a := b]$ ) then
        return true
      end if
    end if
  else if  $l \in C$  then
    if ISENTAILED( $C \setminus \{l\}, T'$ ) then
      return true
    end if
  end if
end for
return false

```

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

Theorem 7. *The procedure `PRUNEENTAILED` terminates in $\mathcal{O}(\text{size}(S_c(T)))$. Moreover, `PRUNEENTAILED`(C, T) is a normal clausal tree and $S_c(\text{PRUNEENTAILED}(C, T))$ contains exactly the clauses $D \in S_c(T)$ such that $C \not\leq_{eq} D$.*

Algorithm 2 `PRUNEENTAILED`(C, T)

```

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

```

The presented techniques can be extended straightforwardly to store ground flat c-clauses instead of ordinary clauses: it suffices to insert a second index at every leaf in order to store the constraints. Both data-structures are handled similarly, the only difference is that a comparison must be added to verify that the clausal part of the subsuming c-clause is indeed smaller than the subsumed one.

6 Conclusion

Although the superposition calculus is not deductive-complete in general, we have shown that it can be adapted in order to make it able to generate all implicates defined over a given *finite* set of *ground* terms denoted by constant symbols. Furthermore, this is done in such a way that the usual termination properties of the calculus are preserved. By duality, the procedure can be used to generate abductive explanations of first-order formulas.

Our calculus shares some similarities with the constrained superposition calculi of [5, 1, 6]. However in our case the constraint and clausal parts are not defined over disjoint signatures: in contrast the Assertion rules allow one to transfer literals from the clausal part to the constraints. In [5, 1, 6] the constraints are used to store formulas that cannot be handled by the superposition calculus, whereas in our case they are used to store properties that are *asserted* instead of being proved.

A drawback with this calculus is that the user has to explicitly declare the set of abducible terms (i.e., the constants in $\mathcal{A}$). This set must be finite and must contain built-in constants (such as `true` or `0`). Note that, thanks to Theorem 2, unsatisfiable or irrelevant implicates (such as $0 \simeq 1$) can be easily detected and discarded on the fly during proof search. Handling infinite (but recursive) sets of terms is possible from a theoretical point of view: it suffices to add an inference rule generating clauses of the form $a \simeq t$, where t is an abducible ground terms and a is a fresh constant symbol. It is straightforward to check that completeness is preserved, but of course termination is lost. A way to recover termination is to develop additional techniques to restrict the application of this rule by selecting the terms t . This could be done either statically, from the initial set of clauses, or dynamically, from the information deduced during proof search. Roughly speaking, the addition of an equation $a \simeq t$ is useless if it cannot generate $\mathcal{A}$ -flat-clauses, and this could be proven automatically at least in some particular cases. Using existing calculi for *symbolically* computing sets of consequences of a given schema of clauses could be used for this purpose (see, e.g., [23]). Such a criterion could be useful also if the considered set of terms is finite, since it can reduce the search space.

Another possible extension is to assume that the set of abducibles is itself defined by a first-order formula (provided by the user). In this case, proving that a term is abducible becomes part of the derivation generating the corresponding implicate.

References

1. E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic sup(la) . In S. Ghilardi and R. Sebastiani, editors, *FroCoS 2009*, volume 5749 of *LNCS*, pages 84–99. Springer, 2009.
2. A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 10(1):129–179, January 2009.
3. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
4. L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.
5. L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3):193–212, 1994.

6. P. Baumgartner and U. Waldmann. Hierarchic superposition with weak abstraction. In M. P. Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2013.
7. R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*, volume 31 of *Applied Logic Series*. Kluwer Academic Publishers, 2004.
8. T. Castell. Computation of prime implicants and prime implicants by a variant of the davis and putnam procedure. In *ICTAI*, pages 428–429, 1996.
9. J. De Kleer. An improved incremental algorithm for generating prime implicants. In *Proceedings of the National Conference on Artificial Intelligence*, pages 780–780. John Wiley & Sons Ltd, 1992.
10. I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In R. Cousot and M. Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2010.
11. I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken. Minimum satisfying assignments for smt. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 394–409. Springer, 2012.
12. M. Echenim and N. Peltier. A Calculus for Generating Ground Explanations. Technical report, CoRR, abs/1201.5954, 2012.
13. M. Echenim and N. Peltier. A Calculus for Generating Ground Explanations. In *Proceedings of the International Joint Conference on Automated Reasoning (IJ-CAR’12)*, volume 7364, pages 194–209. Springer LNCS, 2012.
14. M. Echenim and N. Peltier. A Superposition Calculus for Abductive Reasoning. Technical report, CoRR, abs/1406.0303, 2014.
15. M. Echenim, N. Peltier, and S. Tourret. An approach to abductive reasoning in equational logic. In *Proceedings of IJCAI’13 (International Conference on Artificial Intelligence)*, pages 3–9. AAAI, 2013.
16. M. Echenim, N. Peltier, and S. Tourret. An Approach to Abductive Reasoning in Equational Logic (long version). Technical report, LiG, 2013. <http://membres-lig.imag.fr/peltier/EPT13.pdf>.
17. E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.
18. L. Henocque. The prime normal form of boolean formulas. *Technical report at <http://www.Isis.org/fiche.php>*, 2002.
19. P. Jackson and J. Pais. Computing prime implicants. In *10th International Conference on Automated Deduction*, pages 543–557. Springer, 1990.
20. A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2):185–206, 1990.
21. E. Knill, P. Cox, and T. Pietrzykowski. Equality and abductive residua for horn clauses. *Theoretical Computer Science*, 120:1–44, 1992.
22. C. Lynch and B. Morawska. Automatic Decidability. In *Proc. of 17th IEEE Symposium on Logic in Computer Science (LICS’2002)*, pages 7–16, Copenhagen, Denmark, July 2002. IEEE Computer Society.
23. C. Lynch, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic decidability and combinability. *Information and Computation*, 209(7):1026 – 1047, 2011.
24. P. Marquis. Extending abduction from propositional to first-order logic. In P. Jorrand and J. Kelemen, editors, *FAIR*, volume 535 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 1991.
25. A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 250–264, 2009.

26. A. Matusiewicz, N. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime impicates. *Foundations of Intelligent Systems*, pages 203–213, 2011.
27. M. C. Mayer and F. Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of the IGPL*, 1(1):99–117, 1993.
28. O. Meir and O. Strichman. Yet another decision procedure for equality logic. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV’05*, pages 307–320, Berlin, Heidelberg, 2005. Springer-Verlag.
29. H. Nabeshima, K. Iwanuma, and K. Inoue. Solar: A consequence finding system for advanced reasoning. In M. C. Mayer and F. Pirri, editors, *TABLEAUX*, volume 2796 of *Lecture Notes in Computer Science*, pages 257–263. Springer, 2003.
30. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
31. L. Simon and A. Del Val. Efficient consequence finding. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 359–370, 2001.
32. P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *Electronic Computers, IEEE Transactions on*, 4:446–456, 1967.