A Rewriting Strategy to Generate Prime Implicates in Equational Logic

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

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

Abstract. Generating the prime implicates of a formula consists in finding its most general consequences. This has many fields of application in automated reasoning, like planning and diagnosis, and although the subject has been extensively studied (and still is) in propositional logic, very few have approached the problem in more expressive logics because of its intrinsic complexity. This paper presents one such approach for flat ground equational logic. Aiming at efficiency, it intertwines an existing method to generate all prime implicates of a formula with a rewriting technique that uses atomic equations to simplify the problem by removing constants during the search. The soundness, completeness and termination of the algorithm are proven. The algorithm has been implemented and an experimental analysis is provided.

1 Introduction

The automated generation of the prime implicates of a formula (i.e. its most general consequences, shortened as p.i. from this point on) has been a topic of interest in automated reasoning because of its various applications (e.g. program analysis, knowledge representation...). Generating the p.i. of a formula allows one to extract relevant information, and is useful for instance to remove redundant variables, to simplify the formula, to identify sufficient conditions, etc. The notion of p.i. and their duals, prime implicants, were first introduced for propositional logic in 1955 [21] and from that point on, a lot of algorithms were designed for their computation. Efficient algorithms for computing p.i. in propositional logic use either variants of the resolution rule [5, 12, 13, 24] or decomposition-based approaches in the spirit of the DPLL method [3, 4, 10, 11, 15, 17, 22, 23]. However, most applications of automated reasoning (e.g. in program verification) require the handling of properties and theories that cannot be expressed in propositional logic, hence the need to extend tools such as p.i. generators to more expressive logics (such as quantifier-free equational logic or first-order logic). One of the strong points of the decomposition methods is that they can be applied to all kinds of formulæ, while resolution-based methods can only deal with formulæ in clausal normal form. On the other hand, decomposition methods are designed to handle only finitely many propositional variables, which greatly

impairs the possibility of extending such algorithms to more expressive logics. However, extending resolution-based methods is not a trivial task either, since crucial characteristics such as completeness or termination of propositional algorithms are lost in more expressive logics, rendering them useless. Thus, such an extension must be designed very carefully, which explains why a domain that was extensively studied over the past sixty years contains so few results outside of propositional logic. Nevertheless, methods have been devised for generating p.i. in first-order logic, based mostly either on first-order resolution [16] or the sequent calculus [18]. These methods can handle equational reasoning, the domain targeted in this paper, by adding equality axioms. Other techniques also exist, such as [14] which proposes a built-in method for handling equational reasoning based on an analysis of unification failures. These approaches are very general but not well-suited for real-world applications since termination is not ensured (even for standard decidable classes); furthermore they include no technique for removing equational redundancies, which is a major source of inefficiency. [27] uses the superposition calculus [1] to generate *positive* and *unit* p.i. for specific theories. However, as shown in [9, 14], the superposition calculus is not complete for non-positive or non-unit p.i. Some work has also been done on domains near first-order logic: [2] focuses on a modal logic and presents an extensive study of p.i. in this context. [7] devises a technique for computing some specific prime implicants called *minimum satisfying assignments* in several theories, provided there exists a decision procedure for testing the satisfiability of first-order formulæ in the considered theory (e.g. Presburger arithmetic). This technique is applied in [6] to perform a semi-automated bug detection. [25, 26] propose an approach to synthesize p.i.-like constraints ensuring that a system satisfies some invariant or safety properties. This approach relies on external provers to check satisfiability of first-order formulæ in some base theories. It is very generic and modular, however no automated method is presented to simplify the obtained constraints.

This paper focuses on the generation of p.i. in function-free equational logic: the considered formulæ are boolean combinations of equations between constants. This research stems from the design of a method for abductive reasoning in first-order logic [8], in which a superposition-based calculus is devised to generate function-free consequences of first-order formulæ. This superposition procedure is sound and deduction-complete (for ground function-free implicates; the absence of function symbols is not really restrictive, since functions can be reduced to equalities by adding substitutivity axioms [20]) but the obtained formulæ contain many redundancies which make them hard to analyze. The automated generation of their p.i. allows for the elimination of these redundancies and the computation of a minimal representation of the given formulæ. Note that in [25] a similar lack of parsimony is also identified as one of the main issues. In [9], we designed a tool that is capable of efficiently finding the most general consequences of a quantifier-free equational formula with no function symbols. The proposed algorithm is somewhat similar to the resolution-based p.i. generation method for propositional logic of [5] in its structure, but uses built-in techniques

to handle the properties of the equality predicate. This affects both the representation of clauses, i.e. the way they are stored and tested for redundancy, and their generation: instead of using the resolution method, new inference rules that can be viewed as a form of relaxed paramodulation are defined. An implementation of this algorithm, named Kparam, was compared to state-of-the-art propositional p.i. generation tools by respectively feeding in ground equational formulæ and equivalent propositional abstractions. The Kparam tool outperforms the propositional one in most cases, but it performs badly on some problems. A careful analysis of the experimental results has shown that this is due, for a large part, to the lack of an efficient technique for handling equational simplifications. Obviously, a most natural and efficient way of handling an equation $a \simeq b$ is to uniformly replace one of the terms, say a, by the other, b, thus yielding a simpler problem. It is clear that this operation preserves satisfiability, because a formula $F \wedge a \simeq b$ is satisfiable iff $F[b/a]$ is. The application of such a strategy in the context of p.i. generation raises two important and related issues. First, how to reconstruct the set of implicates of the original problem $F \wedge a \simeq b$, from that of $F[b/a]$? This is not obvious, since, although rewriting preserves satisfiability, it does not in general preserve the set of implicates, as shown later. Second, how to intertwine the systematic application of the rewriting operation with the overall algorithm used to handle the clauses incrementally? This last point is important because the equational simplifications are not transparent to the overall process of p.i. generation. Adjustments are needed, that obviously should not counterbalance the gain of handling equations. In this paper, we investigate both issues and provide solutions for each of them, yielding a much more efficient algorithm for generating implicates of ground equational formulæ. This algorithm is proved to be sound, terminating and complete, thus generating all implicates of the input up to redundancy in a finite time. Experimental comparisons show that equational propagation improves the performances of the algorithm by several orders of magnitude.

Structure of the paper. In Sect. 2 the original strategy from [9] is introduced along with the notations necessary to follow the technical part of the article. Section 3 is a presentation of the rewriting algorithm used in the first step of the strategy and the theoretical properties (completeness and termination) of the global algorithms are provided in Sect. 4. An experimental comparison is conducted in Sect. 5 and the final section contains a summary of the obtained results, along with some lines of future work. Due to space restriction, the proofs are omitted (a technical report containing the proofs is available on the authors web pages).

2 On Equational Logic and Prime Implicate Generation

This section contains the necessary definitions about equational logic along with a simplified presentation of the starting point of our work, namely the p.i. generation algorithm of [9].

2.1 Equational Logic

Let Σ be a finite set of *constants* denoted by a,b,c... We assume a total order $\prec$ on Σ. We also write $a \succ b$ if $b \prec a$. A *literal* l is either an *atom* (or *equation*) $a \simeq b$ (where $a, b \in \Sigma$ and $\simeq$ is the symbol for semantic equality), or the negation of an atom (or *disequation*) $a \not\simeq b$. A literal written $a \bowtie b$ denotes either $a \simeq b$ or $a \not\simeq b$ and by commutativity $a \bowtie b$ and $b \bowtie a$ are considered equivalent. The literal l^c denotes the complement of l, i.e. $a \not\simeq b$ if $l = a \simeq b$ and $a \simeq b$ if $l = a \not\simeq b$. A *clause* C is a disjunction (or multiset) of literals. C^+ is the clause composed of the atoms in C and C^- is composed of the disequations in C. A clause is *positive* if $C^- = \emptyset$. The empty clause is denoted by $\Box$ and $|C|$ is the number of literals in C. An *atomic clause* is a positive unit clause. A *formula* S is a set of clauses. For every clause C, $\neg C$ denotes the formula $\{\{l^c\} \mid l \in C\}$.

An *equational interpretation* $\mathcal{I}$ is a partition of Σ into equivalence classes. Given two constant symbols a and b, we write $a =_\mathcal{I} b$ if a and b belong to the same equivalence class in $\mathcal{I}$, and in this case we say that $a \simeq b$ is *true* in $\mathcal{I}$ (respectively, if $a \neq_\mathcal{I} b$ then $a \not\simeq b$ is true in $\mathcal{I}$). This notation is extended to literals: $a \bowtie b =_\mathcal{I} c \bowtie d$ means that both literals have the same sign and that either $a =_\mathcal{I} c$ and $b =_\mathcal{I} d$ or $a =_\mathcal{I} d$ and $b =_\mathcal{I} c$ (this implies that both literals have the same truth value in $\mathcal{I}$, but the converse does not hold). A clause C is true in $\mathcal{I}$ if C contains at least one literal that is true in $\mathcal{I}$ and a formula S is true in $\mathcal{I}$ if all clauses in S are true in $\mathcal{I}$. Let E represent either a literal, a clause or a formula, then $\mathcal{I} \models E$ means that E is true in $\mathcal{I}$ and in this case $\mathcal{I}$ is called a *model* of E. The notation $E \models E'$ means that all the models of E are also models of E'. If $E \models E'$ and $E' \models E$ then we write $E \equiv E'$. A *tautology* is a clause that is true in all equational interpretations. Unless stated otherwise, only non-tautological clauses will be considered. A *contradiction*, e.g. $\Box$ or $a \not\simeq a$, is a clause with no model. To each clause C we associate a special interpretation $\mathcal{I}_C$ such that $a =_{\mathcal{I}_C} b$ iff $\neg C \models a \simeq b$. To lighten notations we write $a =_C b$ instead of $a =_{\mathcal{I}_C} b$. Note that $\mathcal{I}_C \models C$ iff C is a tautology. The following related notations are also used: $[a]_C \stackrel{\text{def}}{=} \{b \in \Sigma \mid a =_C b\}$ is the equivalence class of a in $\mathcal{I}_C$ and $a_{\downarrow C} \stackrel{\text{def}}{=} \min_\succ [a]_C$ is the *representative* of the class $[a]_C$.

In the original method of p.i. generation from [9], a critical point is redundancy detection. To deal with the constraints induced by the equality axioms, we define a redundancy criterion named *eq-subsumption*, essentially equivalent to semantic entailment.

Definition 1. *Let C, D be two clauses. The clause D eq-*subsumes *C (written $D \leq_{eq} C$) iff the two following conditions hold:*

- *for all $a, b \in \Sigma$, if $\neg D \models a \simeq b$ then $\neg C \models a \simeq b$;*
- *for every positive literal $l \in D$, there exists a literal $l' \in C$ such that $l =_C l'$.*

$D <_{eq} C$ means that $D \leq_{eq} C$ and $C \not\leq_{eq} D$. If S,S' are formulæ, we write $S \leq_{eq} C$ if $\exists D \in S$, $D \leq_{eq} C$ and $S \leq_{eq} S'$ if $\forall C \in S'$, $S \leq_{eq} C$. A clause C is redundant *in S if either C is a tautology or there exists a clause $D \in S$ such that $D <_{eq} C$. A clause set S is* subsumption-minimal *if it contains no redundant clause, i.e. $\forall C \in S$, C is not redundant in S.*

Example 2. Let $C = a \not\simeq b \vee b \not\simeq c \vee a \simeq d$ and $D = a \not\simeq c \vee b \simeq d$. Then $\mathcal{I}_C = \{\{a,b,c\},\{d\}\}$ and $\mathcal{I}_D = \{\{a,c\},\{b\},\{d\}\}$, thus $D \leq_{eq} C$ because $\{a,c\} \subseteq \{a,b,c\}$ and $a \simeq d =_C c \simeq d$. On the other hand $a \not\simeq e \vee a \simeq d \not\leq_{eq} C$ and $a \not\simeq b \vee b \simeq e \not\leq_{eq} C$, respectively because $\{a,e\} \not\subseteq \{a,b,c\}$ and because $b \simeq e \neq_C a \simeq d$.

This criterion offers a syntactic method to detect equational entailment.

Theorem 3. (Th. 8 of [9]) *Let C and D be two clauses. If C is not a tautology then $D \models C$ iff $D \leq_{eq} C$.*

To reduce the work of the redundancy detection algorithms, a normal form is defined for clauses which projects all equivalent clauses onto a single one, thus drastically reducing the number of clauses to be considered.

Definition 4. *The normal form of a non-tautological clause C is:*

$$C_{\downarrow} \stackrel{def}{=} \left(\bigvee_{a \in \Sigma, a \neq a_{\downarrow C}} a \not\simeq a_{\downarrow C} \right) \vee \left(\bigvee_{a \simeq b \in C} a_{\downarrow C} \simeq b_{\downarrow C} \right)$$

and all the literals in $C_{\downarrow}$ occur only once. A formula S is in normal form (denoted by $S_{\downarrow}$) iff all its non-tautological clauses are in normal form. The normal form of a set of atomic clauses $U = \{a_i \simeq b_i\}_{i \in \{1 \ldots n\}}$ is the set of atomic clauses $U_{\downarrow} \stackrel{def}{=} \left\{a'_j \simeq b'_j\right\}_{j \in \{1 \ldots m\}}$ such that $(\bigvee_{i=1}^{n} a_i \not\simeq b_i)_{\downarrow} = \bigvee_{j=1}^{m} a'_j \not\simeq b'_j$.

Proposition 5. (Prop. 4 of [9]) *For every non-tautological clause C, $C_{\downarrow}$ is equivalent to C. Furthermore, if D and C are equivalent and non-tautological then $C_{\downarrow} = D_{\downarrow}$.*

Example 6. Let $C = a \not\simeq b \vee b \not\simeq c \vee a \not\simeq c \vee b \simeq d$. If $a \succ b \succ c \succ d$ then the normal form of C is $C_{\downarrow} = b \not\simeq c \vee a \not\simeq c \vee c \simeq d$.

In all algorithms, we assume that the formulæ are always subsumption-minimal. If a formula S is described by set operations, the redundant clauses are automatically removed. The process is straightforward for all operations except the difference operation, which is defined as follows: for two formulæ S_1 and S_2, $S_1 \backslash S_2 \stackrel{\text{def}}{=} \{C \in S_1 \mid \forall D \in S_2, D \not\models C\}$. Note that if $S_2 \subseteq S_1$ then, since S_1 is subsumption-minimal, only clauses actually belonging to S_2 are removed from S_1. The subject of clause manipulations is not developed further in this article since it is not essential for the understanding of the present paper and there is no significant change in their use w.r.t. the algorithm described in [9].

2.2 Implicate Generation

Definition 7. *A clause C is an* implicate *of a formula S if $S \models C$. An implicate C is a* prime implicate *of S if C is not a tautology, and for every clause D such that $S \models D$, either $D \not\models C$ or $C \models D$. Given a formula S, $PI(S)$ is the set of all the p.i. of S.*

Intuitively, a p.i. of a formula is a consequence that is as general as possible. If any information is removed from it, it is no longer a consequence of the original formula.

Example 8. Let $S = \{a \simeq b \vee d \simeq e, a \simeq c, d \not\simeq e\}$. Both $c \simeq b$ and $c \simeq b \vee d \simeq e$ are implicates of S, but only $c \simeq b$ is a p.i., because $c \simeq b <_{eq} c \simeq b \vee d \simeq e$.

A standard method for testing the satisfiability of equational clause sets is the superposition calculus. This calculus is refutationally complete, meaning that it generates a contradiction from every unsatisfiable set. It is however not complete for deduction, since we may have $S \models C$ even if C cannot be generated from S.

Example 9. Consider $S = \{a \simeq b, c \not\simeq d \vee a \not\simeq d\}$ with $a \succ b \succ c \succ d$. By superposition, the only new implicate that can be generated is $c \not\simeq d \vee b \not\simeq d$ but there are other implicates of S such as $b \not\simeq c \vee a \not\simeq d$.

The implicates of an equational formula are generated using a relaxed paramodulation calculus that permits the replacement of arbitrary constants (instead of identical ones), by adding equality conditions in the resulting clause. For example, the paramodulation rule usually applies between a clause $C[a]$ (a clause C containing the constant a) and $a \simeq b \vee D$, yielding the clause $C[b] \vee D$. In our setting, the clauses $C[a']$ where $a \neq a'$ and $a \simeq b \vee D$ generate $a \not\simeq a' \vee C[b] \vee D$ which can be understood as "if $a \simeq a'$ holds then so does $C[b] \vee D$". Formally, the following rules define the so-called $\mathcal{K}$-paramodulation calculus.

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

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

$$\text{Negative Multi-Paramodulation (M):} \quad \frac{\bigvee_{i=1}^{n}(a_i \not\simeq b_i) \vee P_1 \qquad c \simeq d \vee P_2}{\bigvee_{i=1}^{n}(a_i \not\simeq c \vee d \not\simeq b_i) \vee P_1 \vee P_2}$$

The rule `M` can be applied to one or several disequations at once.

Example 10. Let $C = a \not\simeq d \vee c \not\simeq e \vee d \simeq e$ and $D = a \simeq b$. Using `M`, it is possible to generate the clause $(a \not\simeq a \vee) b \not\simeq d \vee c \not\simeq e \vee d \simeq e$ by selecting only $a \not\simeq d$ in C and the clause $(a \not\simeq a \vee) b \not\simeq d \vee a \not\simeq c \vee b \not\simeq e \vee d \simeq e$ by selecting both $a \not\simeq d$ and $c \not\simeq e$.

Theorem 11. (Th. 13 of [9]) *The $\mathcal{K}$-paramodulation calculus is complete for deduction, i.e. it generates all the implicates of any formula up to redundancy.*

A formula S is *saturated up to redundancy* iff all clauses that can be inferred from premises in S using the rules `P`, `F` or `M` are either in S or redundant w.r.t. S. Note that unlike the superposition calculus, no ordering restrictions are imposed on the premises. This is needed for completeness, as shown in the following example.

Example 12. Let $C = b \simeq c$ and $D = a \simeq c$, with $a \succ b \succ c$. Usual ordering restrictions prevent the generation of the clause $a \simeq b$ because c, being the smallest constant, cannot be replaced by b.

In the algorithms, the following additional notation related to the $\mathcal{K}$-paramodulation calculus is needed.

Definition 13. *Let S be a formula, and C be a clause. $S_{\vdash i,C}$ is the set of all clauses obtained from $S \cup \{C\}$ by exactly i steps of $\mathcal{K}$-paramodulation such that at least one parent of each $\mathcal{K}$-paramodulation step is C. Similarly, we denote by $S_{\vdash C}$ the set of all clauses (up to redundancy) generated by any number of $\mathcal{K}$-paramodulation steps from $S \cup \{C\}$ where C is always one of the parents.*

Algorithm 1 KPARAM(S)

$T := \emptyset$
$S_1 := S$
while $S_1 \neq \emptyset$ **do**
  Choose a clause $C \in S_1$
  if $T \not\leq_{eq} C$ **then**
    $T := T \cup \{C\} \setminus \{D \in T \mid C \leq_{eq} D\}$
    $S_1 := (S_1 \cup T_{\vdash 1,C}) \setminus \{C\}$
  else
    $S_1 := S_1 \setminus \{C\}$
  end if
end while
return T

To generate only the p.i. of a formula, the redundant implicates must be deleted as soon as possible to avoid using them to generate other redundant implicates. For this purpose, the algorithm KPARAM (Algorithm 1, originally proposed in [9]) selects implicates one at a time from a *waiting set* S_1 and uses the selected clause in $\mathcal{K}$-paramodulation inferences with previously selected clauses to generate new implicates. The newly generated clauses are stored in the waiting set and a new implicate can then be selected. Redundant clauses found during the process are removed. The non-redundant used clauses are stored in the *processed set* T[1]. This procedure was proved to be sound, complete and terminating.

3 Atomic Rewriting

To improve the performance of the algorithm described in Sect. 2, we incorporate a rewriting strategy, atomic rewriting (otherwise known as equational simplification), to the process of implicate generation. It simplifies the problem by reducing the number of constants it contains. The underlying principle is simple: assume that an atomic clause $a \simeq b$ is an implicate of a formula S. It is clear that for every model $\mathcal{M}$ of S, necessarily $a =_{\mathcal{M}} b$. Since a and b are always equal, they can be substituted with each other and it is actually possible

[1] KPARAM is an instance of the *given clause* algorithm in the *Otter* variant [19].

to entirely replace one of these constants with the other in the formula, storing the atom $a \simeq b$ apart to avoid any loss of information. Note that the removal of an atom can lead to the generation of new ones as shown in the following example.

Example 14. Consider the formula $S = \{a \simeq b, a \not\simeq b \vee c \not\simeq d, a \not\simeq c \vee a \simeq e, b \not\simeq c \vee b \simeq e, c \simeq a \vee c \simeq b\}$. Using the atom $a \simeq b$, the formula S can be rewritten into $S' = \{b \simeq b, b \not\simeq b \vee c \not\simeq d, b \not\simeq c \vee b \simeq e, b \not\simeq c \vee b \simeq e, c \simeq b \vee c \simeq b\}$ and further simplified into $S'' = \{c \not\simeq d, b \not\simeq c \vee b \simeq e, c \simeq b\}$. Since S'' contains the atom $c \simeq b$, it can in turn be rewritten in $S^{(3)} = \{c \not\simeq d, b \not\simeq b \vee b \simeq e\}$, etc., until only $c \not\simeq d$ remains in the formula.

We introduce the following notations:

Notation 15 *Let S be a formula and U be a set of atomic clauses:*

- *for a and b constants with $a \succ b$, $S[b/a]$ is the formula S where every occurrence of a is replaced by b,*
- *$S[U]$ is the set S where every constant a is replaced by $\min\{a' \,|\, U \models a \simeq a'\}$. For example, if U contains $a \simeq b$ and $a \simeq c$ with $a \succ b \succ c$ then both a and b are replaced by c in $S[U]$.*

In what follows, we will invoke a procedure atomRewrite that recursively removes the atomic clauses appearing in a formula and rewrites all the remaining clauses according to the clauses extracted, until no atom remains. atomRewrite(S) returns the pair $\langle S', U\rangle$ made of the rewritten formula S' and a set U of extracted atomic clauses such that $S \models U$, and $S' = S[U]$ where S' contains no atomic clause. Note that U does not necessarily contain all the atomic clauses that are logical consequences of S. However, it necessarily contains all those occurring in S and those generated by atomic rewriting.

Example 16. Assume that $S = \{a \simeq b, a \not\simeq b \vee c \not\simeq d, c \simeq d \vee e \simeq f\}$. Then invoking atomRewrite(S) returns $S' = \{c \not\simeq d, c \simeq d \vee e \simeq f\}$ and $U = \{a \simeq b\}$, even though it is simple to verify that $S \models e \simeq f$.

New "hidden" atomic implicates like the one in the previous example can be generated at any iteration of the strategy, hence the atomic rewriting should be applicable not only on the initial clause set but also on the newly generated clauses. However, in order to preserve completeness, some clauses occurring in the processed set must then be transferred back to the waiting set (i.e. resp. T_1 and S_1 in Algorithms 1 and 2), otherwise some inferences involving the rewritten clauses of the processed set can never occur. A straightforward way to ensure completeness would be to transfer all clauses back to the waiting set, but this yields a very inefficient algorithm. The following definition introduces a refined criterion that strongly reduces the number of clauses that must be reprocessed.

Definition 17. *A clause D is $\langle a, b\rangle$-*neutral *if $D^+[b/a] = (D[b/a]_\downarrow)^+$. The function* Neutral$(D, a, b)$ *returns true iff D is $\langle a, b\rangle$-neutral.*

This property means that the replacement of a by b does not affect the representatives of the equivalence classes occurring in the positive part of a clause, even if it contains both a and b.

Example 18. Consider the clauses $C = a \not\simeq c \vee b \not\simeq d \vee c \simeq e$ and $D = a \not\simeq c \vee b \not\simeq c \vee c \simeq d$, with $a \succ b \succ c \succ d$. C is not $\langle a, b\rangle$-neutral since $C^+[b/a] = c \simeq e$ while $(C[b/a]_\downarrow)^+ = d \simeq e$. On the other hand D is $\langle a, b\rangle$-neutral because $D^+[b/a] = (D[b/a]_\downarrow)^+ = c \simeq d$.

In the procedure splitAtomRewrite (Algorithm 2) we therefore assume that every non-$\langle a, b\rangle$-neutral clause occurring in T_1 is transferred back to S_1 after rewriting. This procedure takes as an input a pair (T, S) of processed set/waiting set, and returns the new sets after rewriting every atomic clause they contain. $\langle a, b\rangle$-neutrality is the key ensuring the completeness of the algorithm. The informal and intuitive justification is that, if C is $\langle a, b\rangle$-neutral, then all inferences that can be performed on the clause $C[b/a]$ can be "simulated" by inferences with descendants of C. Hence the clause $C[b/a]$ does not need to be considered again.

Algorithm 2 splitAtomRewrite(T, S)

```
U_1 := {a ≃ b ∈ T ∪ S}↓
T_1 := T                                   // T_1 is the processed set
S_1 := S                                   // S_1 is the waiting set
U := ∅
while U_1 ≠ ∅ do
    extract a clause a ≃ b from U_1 and put it into U
    T_2 := {D | ∃D' ∈ T_1, D = (D'[b/a])↓ ∧ Neutral(D', a, b)}
    S_2 := (S_1[b/a])↓ ∪ {(T_1[b/a])↓ \ T_2}
    U_1 := (U_1 ∪ {u ≃ v ∈ T_2 ∪ S_2})↓
    T_1 := T_2
    S_1 := S_2
end while
return ⟨T_1, S_1, U⟩
```

When initializing U_1, taking atoms directly from S is possible because every unit implicate of a non-contradictory formula is one of its p.i., since no clause other than $\square$ and itself subsumes it. Note that the replacement of a by b implicitly deletes the clause $a \simeq b$ from the sets T_2 and S_2.

Lemma 19. splitAtomRewrite *terminates.*

4 Prime Implicate Generation: a New Algorithm

The new algorithm combines $\mathcal{K}$-paramodulation with atomic rewriting to simplify the p.i. computation on the fly. This process is presented in Subsection

4.1 and results in the generation of the set of non-atomic p.i. of the simplified problem together with the set of atomic clauses collected during the search. The recovery of the p.i. of the original formula is described in Subsection 4.2. From this point on, any clause appearing in an algorithm is assumed to be in normal form.

4.1 Integration of the Atomic Rewriting

Algorithm 3 SATURATERW(S)

$\langle S_1, U_1 \rangle :=$ ATOMREWRITE(S)
$T_1 := \emptyset$
while $S_1 \neq \emptyset$ **do**
  Choose a clause $C \in S_1$
  $S_2 := S_1 \setminus \{C\}$
  if $T_1 \not\leq_{eq} C$ **then**
    $T_2 := T_1 \cup \{C\}$
    $R_1 := (T_2)_{\vdash 1, C}$
    $\langle T_3, S_3, U_2 \rangle :=$ SPLITATOMREWRITE($T_2, (S_2 \cup R_1)$)
    $U_1 := U_1 \cup U_2$
    $T_1 := T_3$
    $S_1 := S_3$
  else
    $S_1 := S_2$
  end if
end while
return $\langle T_1, U_1 \rangle$

As can be seen in Algorithm 3, atomic rewritings are added to the original procedure both during the initialization phase, where a call to ATOMREWRITE removes the atomic clauses occurring in the original formula, and at each iteration of the main loop, where SPLITATOMREWRITE is used. SATURATERW(S) returns the pair $\langle T, U \rangle$ where T is the set of clauses eventually obtained by saturation and U is the set of atomic clauses collected during proof search (and deleted from the search space by ATOMREWRITE or SPLITATOMREWRITE).

Lemma 20. *The algorithm* SATURATERW *terminates.*

Theorem 21. *Let S be a formula. If $\langle T, U \rangle =$ SATURATERW(S), then T is saturated up to redundancy and contains no positive unit clauses while U contains only positive unit clauses. Additionally $S \models U$ and $T \equiv S[U]$.*

By Theorem 11, we deduce that T contains all its own p.i. These p.i. are also implicates of S (since $S \models T$), but it is clear that T does not in general contain all the p.i. of S. For instance this set also includes U and all clauses that can be inferred from T and U. Reconstructing the set of p.i. of S is the subject of the next section.

Algorithm 4 COMPUTEPI(T, U)

```
T_1 := T
for all C ∈ U do                          // U is in normal form
    T_1 := T_1 ∪ {C}
    R := (T_1 ⊢1,C)\T_1                    // R contains only newly generated clauses
    while R ≠ ∅ do
        T_1 := T_1 ∪ R
        R := (R ⊢1,C)\T_1
    end while
end for
return T_1
```

4.2 Recovery of the Main Solution

The invocation of SATURATERW on an initial clause set S generates a saturated set of non-atomic clauses T and a normalized set of atomic clauses U. To recover the set of p.i. of S from T and U the principle of COMPUTEPI is to apply the $\mathcal{K}$-paramodulation calculus between the p.i. of T and all atomic clauses in U. In this way, for each atom extracted from S by SATURATERW, COMPUTEPI generates the missing implicates, i.e. those containing the constants that had been previously removed. The essential point (which ensures the efficiency of the approach) is that it is not necessary to apply any inference between the newly generated clauses: only the inferences involving U need to be considered. Formally, what renders COMPUTEPI efficient is the fact that all the implicates of a set of clauses $S \cup \{a \simeq b\}$ (with $a \succ b$) are eq-subsumed by clauses recursively obtained by $\mathcal{K}$-paramodulation between $a \simeq b$ and the p.i. of $S[b/a]$ as stated in Lemma 22.

Lemma 22. *Let S be a formula, $a \simeq b$ be a literal such that $a \succ b$ and $S' = (PI(S[b/a]))_{\vdash a \simeq b}$. Let D be a clause such that $S \cup \{a \simeq b\} \models D$, then $S' \leq_{eq} D$. Thus $S' \equiv S \cup \{a \simeq b\}$ and S' is saturated up to redundancy.*

Lemma 23. COMPUTEPI *terminates.*

The following theorem states that the proposed algorithm, composed of successive calls to SATURATERW and COMPUTEPI, is complete, i.e., that it computes all the p.i. of the input formula.

Theorem 24. *Let S be a formula, $\langle T, U \rangle$ = SATURATERW(S) and S' = COMPUTEPI($T, U_{\downarrow}$). Then S' is the set of p.i. of S.*

5 Experimental Results

Both KPARAM and KPARAMRW have been implemented in Ocaml[2]. Below is an experimental comparison of both tools. The benchmark is made of a thousand

[2] See `http://membres-lig.imag.fr/tourret/documents/kparam.tgz` for the source code.

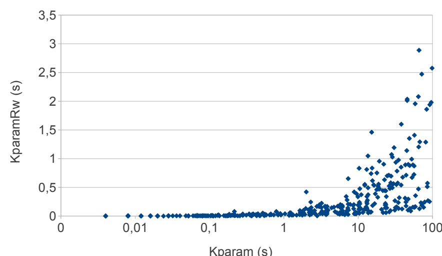

Fig. 1. Execution Time

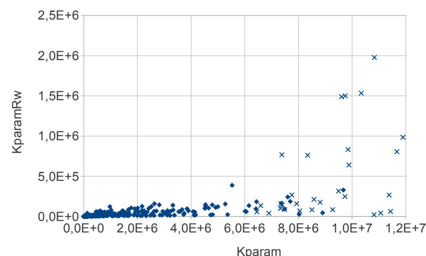

Fig. 2. Generated Implicates

ground flat equational formulæ of a reasonable size[3] that were randomly generated. All tests were conducted on a machine equipped with an Intel core i5-3470 CPU and 4x2 GB of RAM, with a timeout of 100 seconds when not explicitly said otherwise.

A first result worth mentioning is that in KparamRw the execution time of ComputePI is quasi-negligible no matter what the total execution time is: the maximum is less than one second and the mean is 0.09 seconds. In general, it always represents less that 1 percent of the total execution time. Another interesting indicator of the relative superiority of KparamRw compared to Kparam is the fact that while 15% of the benchmark reaches timeout before terminating with Kparam, only 9% does so with KparamRw. An additional 45% of the formulæ have no atomic p.i. and are thus of little interest to us since Kparam and KparamRw merely coincide on such problems. Results concerning the remaining 46% of the benchmark are presented on Fig. 1 & 2. On Fig. 1 the gain of going from Kparam to KparamRw with regards to the execution time can be observed. A logarithmic scale is used for the X axis to highlight that this graph empirically indicates an exponential gain for our benchmark. The results on Fig. 2 were obtained with a timeout of 5 minutes and compare the number of implicates generated by Kparam and KparamRw (for readability issues the scales of the X and Y axis differ). There are two kinds of dots represented on the graph: filled diamonds and X's, the latter representing tests for which Kparam reaches the 5 minutes timeout before terminating. It shows that some problems with atomic p.i. that Kparam cannot solve by computing more than a million implicates can be solved by KparamRw with less than two hundred thousand implicates generated. We also compared our algorithms with `Zres` [24][4], a state-of-the-art tool for p.i. generation in propositional logic that uses a resolution-based algorithm together with ZBDDs for storing clause sets. This system was chosen because it outperforms all other available propositional

[3] Each test is made of 6 clauses with a maximum of 5 literals, using 8 constants. Although the size of the initial formula is small, hundreds of thousand or even millions of implicates are often generated, leading to hundreds of them being eventually kept as prime.

[4] Many thanks to Laurent Simon for providing the executable.

Number of Generated Atoms	0	1	>1	>0	Total
KPARAM	64%	26%	23%	25%	45%
KPARAMRW	64%	83%	80%	82%	73%

Table 1. Percentage of Tests Executed Twice Faster than `Zres`

systems on all our examples. To the best of our knowledge, besides KPARAM no complete p.i. computation tool is available for equational logic[5]. To make the comparison possible, the equational formulæ of the benchmark were translated into equivalent propositional formulæ by abstracting literals away and adding suitable instances of the equality axioms. This straightforward translation is obviously not the most efficient existing method, but it has the advantage of being simple. It still gives a rough execution time reference with which to compare the new algorithm, keeping into account that the time needed for translating the result back to equational logic and removing the redundancies was omitted, so as to underestimate this time. As shown in Table 1, this comparison proved useful by giving an insight of where to improve the original algorithm. The main observation on the line corresponding to KPARAM is that `Zres` is a lot more efficient than this algorithm as soon as atomic implicates appear in the formulæ (only 25% of the tests are faster than `Zres`, while 64% are faster when there are no atomic implicates), which was the motivation for designing KPARAMRW in the first place. As can be seen in the second line of the table, KPARAMRW is a good answer to this problem since an additional 57% of the problems with atomic implicates turn out faster than `Zres` with KPARAMRW, for a total of 82% of these tests being at least twice faster than the state-of-the-art tool. The results also distinguish between formulæ with a single atomic implicate (72%) and several ones (28%). A slight improvement of the performances is noted for the latter, but not as significant as the gap between none and one atomic implicate.

6 Conclusion

In this paper, a new algorithm for the generation of p.i. in ground flat equational logic was presented. It is based on a previous version introduced in [9]. The main idea of this algorithm is to isolate atomic equations to reduce the number of constants handled by the p.i. generator. Although in some applications it may be possible to directly use the simplified results along with the extracted equations, we also devised a way to recover the p.i. of the original input in a efficient way. This new algorithm is terminating, sound and complete and outperforms the previous one when atomic implicates are present. According to our experimental results, the gain is empirically exponential in time. This system can be used

[5] To our knowledge, there exists only one tool, integrated in the Mistral solver [7], that is seemingly similar to KPARAM. However, in contrast to it, the Mistral tool is not complete (it does not compute all the p.i.) hence no comparison is possible.

in connection with the calculus presented in [8] to efficiently generate ground implicates of first-order theories.

An idea to improve atomic rewriting is to find a faster way to generate all the atomic equations entailed by the input formula instead of waiting for them to appear during the inference steps. To do so, the $\mathcal{K}$-paramodulation calculus could be replaced with a more efficient calculus specifically tailored to directly generate all atomic implicates, so that, after a unique rewriting step, the $\mathcal{K}$-paramodulation calculus can be used to generate all remaining non-atomic implicates. To extend further the atomic rewriting strategy, it should be possible to apply it to any equation appearing in the formula in a "divide and conquer" way. Any clause of the form $a \simeq b \vee C$ would then lead to two recursive calls of the strategy, one where $a \simeq b$ is true where the rewriting applies and the other where only C remains. It is still unclear whether this idea is efficient because of two problems: merging the results of the two recursive calls is by no means a simple task, and the fact that there are two calls on formulæ that differ only by one clause may generate a lot of redundant computation steps, thus slowing down the whole process. These questions need a thorough investigation and are one of our objectives for future work.

Up to now, our system has been mainly tested on randomly computed instances. We now plan to apply it, in conjunction with an implementation of the calculus described in [8], to more concrete problems in system verification, particularly for checking properties of algorithms operating on arrays or pointer-based data-structures.

References

1. Bachmair, L. and Ganzinger, H.: Rewrite-based Equational Theorem Proving with Selection and Simplification. In: Journal of Logic and Computation, 1994, vol. 3, no 4, p. 217-247.
2. Bienvenu, M.: Prime implicates and prime implicants in modal logic. In: Proceedings of the National Conference on Artificial Intelligence. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007. p. 379.
3. Bittencourt, G.: Combining syntax and semantics through prime form representation. In: Journal of Logic and Computation, 2008, vol. 18, no 1, p. 13-33.
4. Coudert, O. and Madre, J. C.: A new method to compute prime and essential prime implicants of boolean functions. In: Advanced Research in VLSI and Parallel Systems. Knight and Savage (Eds), 1992, p. 113-128.
5. De Kleer, J.: An improved incremental algorithm for generating prime implicates. In: Proceedings of the tenth National Conference on Artificial Intelligence. AAAI Press, 1992. p. 780-785.
6. Dillig, I., Dillig, T., and Aiken, A.: Automated error diagnosis using abductive inference. In: ACM SIGPLAN Notices. ACM, 2012. p. 181-192.
7. Dillig, I., Dillig, T., McMillan, K. L., and al.: Minimum satisfying assignments for SMT. In: Computer Aided Verification. Springer Berlin Heidelberg, 2012. p. 394-409.
8. Echenim, M. and Peltier, N.: A calculus for generating ground explanations. In: Proceedings of the 6th international joint conference on Automated Reasoning. Springer-Verlag, 2012. p. 194-209.

9. Echenim, M., Peltier, N. and Tourret, S.: An approach to abduction in equational logic. In: Proceeding of the 23d International Joint Conference on Artificial Intelligence. AAAI Press, 2013, p. 531-538
10. Errico, B., Pirri, F., and Pizzuti, C.: Finding prime implicants by minimizing integer programming problems. In: AI-CONFERENCE-. World Scientific Publishing, 1995. p. 355-362.
11. Jackson, P. and Pais, J.: Computing Prime Implicants. In: Proceedings of the 10th International Conference on Automated Deduction. Springer Berlin Heidelberg, 1990. p. 543-557.
12. Jackson, P.: Computing prime implicates incrementally. In: Automated Deduction, CADE-11. Springer Berlin Heidelberg, 1992, p. 253-267.
13. Kean, A. and Tsiknis, G.: An incremental method for generating prime implicants/implicates. In: Journal of Symbolic Computation, 1990, vol. 9, no 2, p. 185-206.
14. Knill, E., Cox, P.T. and Pietrzykowski, T.: Equality and abductive residua for Horn clauses. In: Theoretical Computer Science, 1993, vol. 120, no 1, p. 1-44.
15. Manquinho, V. M., Oliveira, A. L., and Marques-Silva, J.: Models and algorithms for computing minimum-size prime implicants. In: Proceedings of the International Workshop on Boolean Problems. 1998.
16. Marquis, P.: Extending abduction from propositional to first-order logic. In: Fundamentals of artificial intelligence research. Springer Berlin Heidelberg, 1991. p. 141-155.
17. Matusiewicz, A., Murray, N. V., and Rosenthal, E.: Tri-based set operations and selective computation of prime implicates. In: Foundations of Intelligent Systems. Springer Berlin Heidelberg, 2011. p. 203-213.
18. Mayer, M.C. and Pirri, F.: First order abduction via tableau and sequent calculi. In: Logic Journal of IGPL, 1993, vol. 1, no 1, p. 99-117.
19. McCune, W. and Wos, L.: Otter-the CADE-13 competition incarnations. In: Journal of Automated Reasoning, 1997, vol. 18, no 2, p. 211-220.
20. Meir, O. and Strichman, O.: Yet another decision procedure for equality logic. In: Computer Aided Verification. Springer Berlin Heidelberg, 2005. p. 307-320.
21. Quine, W. V.: A way to simplify truth functions. The American Mathematical Monthly, 1955, vol. 62, no 9, p. 627-631.
22. Ramesh, A., Becker, G., and Murray, N. V.: CNF and DNF considered harmful for computing prime implicants/implicates. In: Journal of Automated Reasoning, 1997, vol. 18, no 3, p. 337-356.
23. Rymon, R.: An se-tree-based prime implicant generation algorithm. In: Annals of Mathematics and Artificial Intelligence, 1994, vol. 11, no 1-4, p. 351-365.
24. Simon, L. and Del Val, A.: Efficient consequence finding. In: International Joint Conference on Artificial Intelligence. Lawrence Erlbaum Associates ltd, 2001, vol. 17, p. 359-365.
25. Sofronie-Stokkermans, V.: Hierarchical reasoning for the verification of parametric systems. In: Proceedings of the 5th international conference on Automated Reasoning. Springer-Verlag, 2010. p. 171-187.
26. Sofronie-Stokkermans, V.: Hierarchical reasoning and model generation for the verification of parametric hybrid systems. In: Automated Deduction-CADE-24. Springer Berlin Heidelberg, 2013. p. 360-376.
27. Tran, D., Ringeissen, C., Ranise, S., and al.: Combination of convex theories: Modularity, deduction completeness, and explanation. In: Journal of Symbolic Computation, 2010, vol. 45, no 2, p. 261-286.