A Deductive-Complete Constrained Superposition Calculus for Ground Flat Equational Clauses

M. Echenim, N. Peltier and S. Tourret[1]

Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France `firstname.lastname@imag.fr`

Abstract

We describe an algorithm that generates prime implicates of equational clause sets without variables and function symbols. The procedure is based on constrained superposition rules, where constraints are used to store literals that are asserted as additional axioms (or hypotheses) during the proof search. This approach is sound and deductive-complete, and it is more efficient than previous algorithms based on conditional paramodulation. It is also more flexible in the sense that it allows one to restrict the search space by imposing additional properties that the generated implicates should satisfy (e.g., to ensure relevance).

1 Introduction

An *implicate* of a formula ϕ is a clause that is a logical consequence of ϕ. It is *prime* if it is minimal w.r.t. logical entailment. The computation of prime implicates of logical formulæ (also known as the consequence-finding problem) is a fundamental problem in automated deduction, with many applications in computer science and artificial intelligence. It is essentially useful to compute explanations of observed facts: indeed, by duality, any implicate $l_1 \vee \ldots \vee l_n$ of some formula ϕ in a theory $\mathcal{T}$ corresponds to a conjunction $\neg l_1 \wedge \ldots \wedge \neg l_n$ that logically entails $\neg\phi$ modulo $\mathcal{T}$. This problem has been extensively studied in propositional logic and many algorithms have been proposed to compute implicates efficiently, see, e.g., [3, 8, 9, 12, 13]. These algorithms use either refinements of the Resolution calculus[1] or decomposition procedures in the spirit of the DPLL method. Other approaches have tackled more expressive logics [2, 10, 11, 14, 15]. In [6, 5] we devised an algorithm to generate implicates of ground, function-free equational clause sets, i.e. clause sets built over atoms of the form $a \simeq b$, where a and b are constant The resulting calculus, called $\mathcal{K}$-paramodulation, uses transitivity rules to derive new clauses from premises, together with indexing techniques for efficiently storing generated clauses and deleting redundant ones (modulo logical entailment in equational logic). In particular it is based on a form of "conditional paramodulation", meaning that equality conditions are not checked statically, but *asserted* by adding new disequations to the derived clause. For instance, given a clause $C[a]$ and an equation $a' \simeq b$, our calculus generates the clause $a \not\simeq a' \vee C[b]$, which can be interpreted as $a \simeq a' \Rightarrow C[b]$ (the condition $a \simeq a'$ is asserted). The procedure is sound and deductive-complete (i.e., all prime implicates are generated), and also more efficient than existing approaches based on a reduction from equational (quantifier-free, function-free) logic to propositional logic.

In this paper we propose a new, slightly different approach in which a distinction is drawn between the literals that are asserted and the standard ones – the former being attached to the clauses as constraints. It is clear that from a theoretical point of view this approach can strongly increase the search space, since any clause of size n can now have 2^n distinct equivalent

[1]Unordered Resolution is well-known to be complete for implicates generation.

representatives, depending on which literals are stored in the constraints. However, it also has many advantages:

- First, all the usual ordering restrictions or selection strategies of the superposition calculus [1] can be carried over to the new procedure. This was not the case for our previous algorithm: their addition renders the $\mathcal{K}$-paramodulation calculus incomplete for consequence-finding.
- Second, this approach offers the possibility to control the literals that can be asserted, for instance to limit the number of asserted literals, to impose additional syntactic restrictions on them, or even to test semantic conditions. This is especially important in practice since the number of implicates of a formula is typically huge, as soon as equality axioms are considered.

The principle of the presented calculus is to apply the standard superposition calculus, enriched by new rules that allow the addition of ground unit clauses as new axioms (or hypotheses) during the search. The additional axioms used to derive a given clause are attached to the clause as constraints and these constraints are taken into account when testing redundancy[2]. Once an empty clause has been generated, the negation of the conjunction of the hypotheses used to derive it can be returned as an implicate of the considered clause set.

2 Preliminary Definitions

Let $\mathcal{C}$ be a set of *constant symbols*. An *atom* is an equation $a \simeq b$ where $a, b \in \mathcal{C}$. A *literal* is either an atom (*positive* literal) or the negation of an atom (*negative* literal), written $\neg(a \simeq b)$ or $(a \not\simeq b)$. The literal *complementary* of l is l^c, with $(a \simeq b)^c \stackrel{\text{def}}{=} a \not\simeq b$ and $(a \not\simeq b)^c \stackrel{\text{def}}{=} a \simeq b$. The symbol $\bowtie$ is often used to denote indifferently $\simeq$ or $\not\simeq$. A *clause* is a finite set of literals written as a disjunction. The empty clause is denoted by $\Box$.

Definition 1. *A* constraint *is a (possibly empty) conjunction of literals. A* constrained clause *(or* c-clause*) is a pair $[C \,|\, \mathcal{X}]$ where C is a clause and $\mathcal{X}$ is a constraint. Empty constraints are denoted by $\top$. $[C \,|\, \top]$ is often written simply as C and referred to as a standard clause.*

If $\mathcal{X} = \bigwedge_{i=1}^{n} l_i$ then $\mathcal{X}^c$ denotes the clause $\bigvee_{i=1}^{n} l_i^c$. Similarly, if $C = \bigvee_{i=1}^{n} l_i$ then $C^c \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} l_i^c$. We often identify sets of unit clauses with conjunctions, e.g., considering a set of clauses S, we write $S \cup \bigwedge_{i=1}^{n} l_i$ for $S \cup \{l_i \mid i \in [1, n]\}$, and instead of $\{l_1, \dots, l_n\} \subseteq \{l'_1, \dots, l'_m\}$, we write $\bigwedge_{i=1}^{n} l_i \subseteq \bigwedge_{i=1}^{m} l'_i$.

Let $\succ$ be a total ordering on $\mathcal{C}$ (in all examples, we assume that $a \succ b \succ c \succ \dots$). The ordering is extended to atoms, literals and clauses by multiset extension as usual (see, e.g., [1]). A constraint $\mathcal{X}$ (resp. a clause C) is *normalized* if for every equation $a \simeq b$ (resp. disequation $a \not\simeq b$) occurring in $\mathcal{X}$ (resp. C) where $a \succ b$, the constant a occurs only once within $\mathcal{X}$ (resp. C). Let E represent a constraint or a clause, then $E_{\downarrow}$ denotes the normalized form of E. Note that $E_{\downarrow}$ always exists and is unique if E is either a non-tautological clause or a non contradictory constraint.

For any clause C and constraint $\mathcal{X}$, we denote by $C_{\downarrow \mathcal{X}}$ the clause obtained by first replacing every constant x in C by the smallest (w.r.t. $\prec$) constant x' such that $\mathcal{X} \models x \simeq x'$, and then removing all literals of the form $x \not\simeq x$. For example, $(a \simeq b \vee a \not\simeq c)_{\downarrow a \simeq c} = c \simeq b$.

[2]For instance a clause $p \vee q$ with no assumed literals is not necessarily less general than a unit clause p with constraint r.

From a semantic point of view, a constrained clause $[C \mid \mathcal{X}]$ is equivalent to $\mathcal{X}^c \vee C$. For example the c-clause $[c \simeq b \mid a \simeq c \wedge c \not\simeq d]$ is equivalent to $c \simeq b \vee a \not\simeq c \vee c \simeq d$. More specifically, the intended meaning of a c-clause $[C \mid \mathcal{X}]$ is that the clause C can be inferred provided the literals in $\mathcal{X}$ are added as axioms to the considered clause set.

3 A Calculus for Abductive Reasoning

In this section the superposition calculus [1, 17] is extended to sets of c-clauses. Note that the applicability conditions are much simpler than usual ones since c-clauses do not contain variables or function symbols. First, the standard inference rules are extended in a straightforward way by adding the constraints of the premises to the conclusion (cf. Table 1). As usual the calculus is parameterized by the ordering $\succ$ on terms and a selection function s*el*, where s*el*(C) contains all maximal literals in C or (at least) one negative literal. A literal is *selected* in C if it occurs in s*el*(C).

The abduction rules (cf. Table 2) allow for the addition of new hypotheses in the constraint part of a c-clause. To this purpose the most simple solution would be to add to the clause set all tautological axioms of the form $[l \mid l]$ (meaning that l can be derived from l) where l is a ground literal, and then to let the inference rules of Table 1 derive all the consequences of these axioms. However, this solution is not completely satisfactory since there are numerous axioms of the previous form, and that not all of them are relevant w.r.t. the considered clause set. It is preferable to avoid the blind enumeration of axioms, which is why we add rules simulating all possible inferences from these axioms and the already generated c-clauses[3] (cf. Table 2).

We now explain and motivate the form of the new abduction rules. The positive assertion rule asserts an equation $t \simeq u$ as a new hypothesis in the constraint part of a clause. This is done if the addition of such a hypothesis enables the application of the superposition rule into the considered clause. Note that the term u does not necessarily occur in C: the condition is only that it must be strictly smaller than t. The negative assertion rule proceeds in a similar way for disequations, which allow for an application of the superposition rule into them. A literal $t \not\simeq u$ is added to the constraint part of a clause of the form $t \simeq s \vee C$ that is changed into $s \not\simeq u \vee C$, only if this allows a Superposition inference into this new clause (again, the term u does not necessarily occur in the premise).

Example 2. *The following example shows how to derive the implicate* $a \not\simeq c \vee b \simeq d$ *from* $\{a \simeq b, c \simeq d\}$.

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

The negation of $a \simeq c \wedge b \not\simeq d$ *is the desired implicate.*

The usefulness of the c-clause representation becomes apparent when looking at the inference rules (both standard and abduction rules). It is a way to separate the literals that can be used for inferences from the "frozen" ones stored in the constraints, on which no inference should be applied.

[3]From a purely theoretical point of view the two solutions are of course equivalent.

Superposition	$\dfrac{[l \simeq r \vee C \mid \mathcal{X}] \quad [l \bowtie u \vee D \mid \mathcal{Y}]}{[r \bowtie u \vee C \vee D \mid \mathcal{X} \wedge \mathcal{Y}]}$	If $l \succ r$, $l \succ u$, and $(l \simeq r)$ and $(l \bowtie u)$ are selected in $(l \simeq r \vee C)$ and $(l \bowtie u \vee D)$ respectively.
Reflexivity	$\dfrac{[t \not\simeq t \vee C \mid \mathcal{X}]}{[C \mid \mathcal{X}]}$	No condition is imposed.
Factoring	$\dfrac{[t \simeq u \vee t \simeq v \vee C \mid \mathcal{X}]}{[t \simeq v \vee u \not\simeq v \vee C \mid \mathcal{X}]}$	If $t \succ u$, $t \succ v$ and $(t \simeq u)$ is selected in $t \simeq u \vee t \simeq v \vee C$.
Normalization	$\dfrac{[t \not\simeq u \vee t \bowtie v \vee C \mid \mathcal{X}]}{[t \not\simeq u \vee u \bowtie v \vee C \mid \mathcal{X}]}$	If $t \succ u$.

Table 1: Standard Inference Rules

Positive Assertion	$\dfrac{[t \bowtie s \vee C \mid \mathcal{X}]}{[u \bowtie s \vee C \mid \mathcal{X} \wedge t \simeq u]}$	If $t \succ s$, $t \succ u$ and $t \bowtie s$ is selected in $t \bowtie s \vee C$.
Negative Assertion	$\dfrac{[t \simeq s \vee C \mid \mathcal{X}]}{[s \not\simeq u \vee C \mid \mathcal{X} \wedge t \not\simeq u]}$	If $t \succ u$, $t \succ s$, and $t \simeq s$ is selected in $t \simeq s \vee C$.

Table 2: Abduction Rules

3.1 Redundancy Elimination Rule

Redundancy testing is done as usual, except that the constraints must be taken into account; in particular, it is necessary to make sure that the constraints of the redundant c-clause include those of the considered c-clauses.

Definition 3. *A c-clause $[C \mid \mathcal{X}]$ is* redundant *w.r.t. a set of c-clauses S if either $\mathcal{X}$ is unsatisfiable or there exist c-clauses $[D_i \mid \mathcal{Y}_i] \in S$ ($1 \leq i \leq n$) such that $\forall i \in \{1 \dots n\}\, C \succeq D_i$, $\forall i \in \{1 \dots n\}\, \mathcal{Y}_i \subseteq \mathcal{X}$ and $\mathcal{X}', D_1, \dots, D_n \models C$, where $\mathcal{X}'$ denotes the set of literals in $\mathcal{X}$ that are smaller than C.*

The *redundancy elimination rule* removes a c-clause C from a set of c-clauses S (formally $S \cup \{C\} \to S$) if C is redundant w.r.t. $S \setminus \{C\}$. For example, if X is unsatisfiable, then any clause $[C \mid \mathcal{X}]$ is redundant in any set. In practice, we use a stronger notion of redundancy, based on the notion of E-subsumption[4] [5] that is a generalization of the subsumption test to equational clauses:

[4]The E in E-subsumption stands for Equational.

Definition 4. *A clause C E-*subsumes *a clause D (written $C \leq_E D$) iff $C_{\downarrow D^c} \subseteq D_{\downarrow}$. A c-clause $[C \,|\, \mathcal{X}]$ E-*subsumes *a clause $[D \,|\, \mathcal{Y}]$ (written $[C \,|\, \mathcal{X}] \leq_E [D \,|\, \mathcal{Y}]$) iff $C \preceq D$, $C \leq_E D$ and $\mathcal{X} \subseteq \mathcal{Y}$.*

Proposition 5. *If $[C \,|\, \mathcal{X}] \leq_E [D \,|\, \mathcal{Y}]$ then $[D \,|\, \mathcal{Y}]$ is redundant in $\{[C \,|\, \mathcal{X}]\}$.*

Proof. It suffices to remark that if $C \leq_E D$, then $C \models D$ (a formal proof of this fact is provided in [5]). $\square$

Note that both parts of the c-clauses are handled in different ways: the inclusion relation $\subseteq$ used to compare constraints is clearly stronger than the E-subsumption relation $\leq_E$ used for clauses. For instance we have $[a \not\simeq b \vee b \simeq d \mid \top] \leq_E [a \not\simeq c \vee b \not\simeq c \vee c \simeq d \mid \top]$, but $[\square \,|\, a \simeq b \wedge b \not\simeq d] \not\leq_E [\square \,|\, a \simeq c \wedge b \simeq c \wedge c \not\simeq d]$.

Theorem 6. *The constrained calculus is sound and deductive-complete.*

The proof of this theorem is available in the appendix.

Implementation details. Testing redundancy by considering pairs of c-clauses one by one is of course inefficient. In practice, we adopt the following approach. First, c-clauses are normalized by applying the Normalization rule up to irreducibility. It is easy to see that the conclusion is always equivalent to the premise. Since it is also strictly smaller, the premise becomes redundant and can be deleted after the rule is applied. Then the normalized (non-tautological) clauses are stored in an index (called a *clausal tree* in [6, 5]) that is a special type of trie, i.e. a tree with edges labeled by literals. Each branch represents a clause, defined as the disjunction of the literals labeling the edges in the branch. Literals are ordered using the following conventions:

1. In the clausal part, negative literals always occur before positive ones.
2. The other literals are ordered using $\succ$.

Efficient algorithms were devised in [5] to check that a clause is redundant w.r.t. existing clauses in this data-structure, and to remove from the index clauses that are redundant w.r.t. a newly generated clause. The data-structure and algorithms in [6, 5] only handle standard clauses and compare them using the relation $\leq_E$, hence it was necessary to slightly adapt these procedures. This is done as follows. First, a test is added in order to verify that the clausal part of the subsuming c-clause is indeed smaller than the subsumed one. Second, to handle the constraint part of the c-clauses, we insert a trie data-structure at every leaf of the previous index in order to store the constraints of the c-clauses. The above-mentioned algorithms can be combined in a natural way with standard algorithms for membership tests, insertions and updates inside tries (see for instance [7]).

Another index is used in parallel to accelerate the generation of new c-clauses. In this index, the c-clauses are grouped according to the biggest constant in their selected literal and to its sign. For example c-clauses where $a \simeq b$ and $a \simeq c$ are selected with $a \succ b \succ c$, are grouped together while the c-clauses in which $a \not\simeq b$ is selected are stored separately. This permits the recovery of all the clauses that can be used in an inference with a given clause without scanning the whole list. This index allows the overall algorithm to be executed on average twice faster than without it.

4 Restricting the Class of Implicates

The number of implicates of a given formula is usually huge, and it is important in practice to be able to prune the search space by imposing additional restrictions[5] on the implicates that are searched for. This is possible if the considered class of implicates satisfies the following condition.

Definition 7. *A set of constraints $\mathfrak{X}$ is* $\subseteq$-closed *if for every $\mathcal{X} \in \mathfrak{X}$ and constraint $\mathcal{Y}$, $\mathcal{Y} \subseteq \mathcal{X} \Rightarrow \mathcal{Y} \in \mathfrak{X}$. The set $\mathfrak{X}$ is* normalized *if every constraint $\mathcal{X} \in \mathfrak{X}$ is normalized.*

Let $\mathfrak{X}$ be a set of constraints. A set of c-clauses S is *$\mathfrak{X}$-saturated* if every c-clause $[D \,|\, \mathcal{X}]$ such that $\mathcal{X} \in \mathfrak{X}$ that is deducible from S by applying one of the inference rules is redundant w.r.t. S.

Theorem 8. *Let $\mathfrak{X}$ be a normalized and $\subseteq$-closed set of satisfiable constraints. Let S be a set of standard clauses (i.e. c-clauses with empty constraint) and $S^\star$ be a set of clauses obtained from S by applying inference or redundancy deletion rules. If $S^\star$ is $\mathfrak{X}$-saturated and $S \models \mathcal{X}^c$ for some $\mathcal{X} \in \mathfrak{X}$, then there exists $\mathcal{Y} \subseteq \mathcal{X}$ such that $[\square \,|\, \mathcal{Y}] \in S^\star$.*

Remark 9. *The proof of Th. 8 derives from that of Th. 16 (in the Appendix) which is in essence a simplified variant of Th. 8.*

Examples of $\subseteq$-closed sets of constraints include positive (or negative) constraints, or constraints of size at most some fixed $k \in \mathbb{N}$.

Simplification of Equational Clause Sets. For any set of clauses S, the set of constraints $\mathfrak{S}$ such that $\mathcal{X} \in \mathfrak{S}$ if and only if there exists $C \in S$ with $\mathcal{X}^c \models C$, is $\subseteq$-closed. This remark allows us to use our algorithm to efficiently compute a minimal (up to redundancy) equivalent representation of any set of clauses S. This is done as follows. First, S is $\mathfrak{S}$-saturated, and the set I of clauses $\mathcal{X}^c$ such that $[\square \,|\, \mathcal{X}]$ occurs in the saturated set is constructed. By Theorem 8, I is the set of prime implicates of S that occur in $\mathfrak{S}$, i.e., that entail at least one clause in S. Then, for each clause $C \in S$, an implicate $C' \in I$ such that $C' \models C$ is selected[6]. The obtained clause set is equivalent to S and minimal in the sense that all the clauses are minimal w.r.t. logical entailment (in particular no literal can be deleted without affecting the semantics). This simplification method departs from the one described in [4] in which formulæ are reduced by removing literals occurring in them, provided they are useless in the context. For instance the literal l can be removed in $(l \vee \psi) \wedge \phi$ if $\phi, \neg\psi \models l$. Our technique allows for finer simplifications, taking into account equational axioms.

Example 10. *Consider the clause set: $S \overset{def}{=} \{a \not\simeq c \vee b \not\simeq c \vee d \simeq e, a \simeq c \vee a \simeq f, b \simeq c \vee a \simeq f, f \not\simeq b\}$. It is easy to check that $a \not\simeq b \vee d \simeq e$ is an implicate of S and this clause E-subsumes $a \not\simeq c \vee b \not\simeq c \vee d \simeq e$. Our approach computes the clause set $S' \overset{def}{=} \{a \not\simeq b \vee d \simeq e, a \simeq c \vee a \simeq f, b \simeq c \vee a \simeq f, f \not\simeq b\}$ that is equivalent to S and strictly smaller. In contrast, the approach in [4] cannot simplify S since there is no useless literal.*

5 Experiments

There are few prime implicate computation tools available. Most of them are designed for propositional logic only [13, 19] and are not very efficient in solving equational problems (con-

[5]These could be for instance syntactic restrictions.

[6]Such a clause necessarily exists since C itself is an implicate of S – albeit not necessarily minimal.

verted beforehand to propositional logic) [5, 6]. Others are not well suited for comparison with our tool, either for lack of completeness [4] or because they do not handle equality [15]. This is why, to evaluate the efficiency of the constrained calculus (CC), we integrated it in our tool `Kparam` (see [5, 6]) and compared it[7] with the calculus from [6] (Kp) on a thousand of randomly generated benchmarks containing up to 6 clauses of maximum 5 literals made from 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.

These formulæ are written using the TPTP cnf syntax [20]. The results are summarized in Fig.1, where each point represents the intersection of the execution time (logarithmic scale) of a given benchmark by CC (X axis) and by Kp (Y axis). Globally, 56% of the benchmarks run faster using CC. The uppermost diagonal line splits the diagram in two parts, over it are the tests for which CC is at least 10 times faster than Kp. These benchmarks represent roughly 25% of the results. Below the other diagonal are the tests for which Kp is 10 times faster than CC. These benchmarks represent only 5% of the results. This analysis allows us to conclude that CC is more efficient on the considered benchmark. Still depending on the chosen time limit, in case of failure by timeout from CC it is reasonable to try to solve the problem using Kp before extending the computation time or giving up. These results are encouraging, especially given the fact that CC has some features that Kp is lacking (namely the ability to impose restrictions on the implicates).

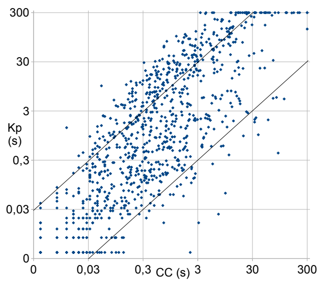

Figure 1: Execution Time Comparison

Testing satisfiability using CC. Some tests were conducted to evaluate the efficiency of CC for satisfiability testing, i.e., when constraining the implicates to size 0. In this case, CC essentially coincides with the standard superposition calculus, except for the normalization step, which is not performed by existing provers (as far as we are aware). Table 3 summarizes the results of a comparison of our algorithm with E [18] on some small unsatisfiable instances of the pigeonhole problem (where there are too many pigeons and not enough holes to put them all) in equational logic. The notation 'pigAxB' indicates that the considered problem involves A pigeons and B holes. Without surprise, E outperforms CC as far as execution time is concerned (this reflects the limits of the our current implementation rather than those of the calculus). Any other result would have been surprising considering that CC is implemented in a purely high-level way, while E has benefited over the years of many carefully designed high- and low-level refinements. The other two columns of Table 3 contain much more satisfying results for CC. They present the number of clauses generated and processed by both systems and in all cases the results are of the same order of magnitude but in favor of CC. This phenomenon is most probably explained by the normalization process used in CC, which prevents it from considering several equivalent clauses. This observation suggests that it could be interesting to integrate a normalization process into the tools dealing with equational logic (this normalization step extends straightforwardly to first-order clauses).

[7]All tests were conducted on a machine equipped with an Intel core i5-3470 CPU and 4x2 GB of RAM, with a timeout of 500 seconds

		execution time	number of clauses generated	number of clauses processed
pig5x4	CC	0.016s	162	74
	E	0.014s	292	79
pig8x7	CC	5.780s	17198	833
	E	0.621s	27248	1218
pig10x9	CC	492.791s	340442	4197
	E	21.153s	542664	7038

Table 3: Comparison between E and the constrained calculus looking for implicates of size 0

5.1 Conclusion

This paper introduced a new and more efficient way of computing the prime implicates of equational formulæ, with the possibility of restricting the results, without loss of the completeness property (for the implicates to which the restriction applies). This feature is useful for example to generate implicates up to a given size, or to compute minimal representations of a given formula (by generating implicates "covering" all possible conjuncts). This work is a first step toward scalable prime implicate generators in equational logic. Possibilities to further extend this work include finding more relevant benchmarks to evaluate the applicability of the calculus to concrete problems and looking for other interesting restrictions preserving the completeness of the calculus. In the longer term, an extension to more expressive logics, either directly or using an approach in the spirit of DPLL(T) [16] will be considered.

References

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

[2] M. Bienvenu. Prime implicates and prime implicants in modal logic. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 379. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

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

[4] 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.

[5] 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.

[6] M. Echenim, N. Peltier, and S. Tourret. A Rewriting Strategy to Generate Prime Implicates in Equational Logic. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'14)*. Springer, 2014.

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

[8] P. Jackson. Computing prime implicates. In *ACM Conference on Computer Science*, pages 65–72, 1992.

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

[10] E. Knill, P. Cox, and T. Pietrzykowski. Equality and abductive residua for horn clauses. *Theoretical Computer Science*, 120:1–44, 1992.

[11] P. Marquis. Extending abduction from propositional to first-order logic. In P. Jorrand and J. Kelemen, editors, *Fundamentals of Artificial Intelligence Research*, volume 535 of *LNCS*, pages 141–155. Springer Berlin, 1991.

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

[13] A. Matusiewicz, N. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. *Foundations of Intelligent Systems*, pages 203–213, 2011.

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

[15] 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.

[16] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL (T). *Journal of the ACM*, 53(6):937–977, 2006.

[17] 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.

[18] S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

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

[20] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

Appendix: Soundness and Completeness

It is easy to check that the previous calculus is sound:

Lemma 11. *Let $[C \,|\, \mathcal{X}]$ be a c-clause derived from n premises $[D_i \,|\, \mathcal{Y}_i]$ with $i \in \{1 \dots n\}$. Then C is a logical consequence of $D_1, \dots, D_n, \mathcal{X}$ and for all i, $\mathcal{Y}_i \subseteq \mathcal{X}$.*

Proof. It is easy to verify that this property holds for each inference rule, the result follows by a straightforward induction on the length of the derivation. □

Lemma 11 permits to deduce the following soundness result:

Corollary 12. *For any c-clause $[C \,|\, \mathcal{X}]$ deducible from a set of clauses S (i.e. c-clauses with empty constraint), C is a logical consequence of $S \cup \mathcal{X}$. In particular, if $C = \Box$ then $S \models \mathcal{X}^c$.*

We now prove that the calculus is deductive-complete, i.e., that it permits to generate every prime implicate of a given set of clauses. The proof relies on the following definitions and proposition.

Definition 13. *For every set of c-clauses S and for every constraint $\mathcal{X}$, we denote by $S|_{\mathcal{X}}$ the set of clauses D (without constraint) such that $[D \,|\, \mathcal{Y}] \in S$ and $\mathcal{Y} \subseteq \mathcal{X}$.*

The following proposition is an immediate consequence of the definition.

Proposition 14. *Let S be a set of c-clauses and let $\mathcal{X}$ be a satisfiable constraint. If a c-clause $[C \,|\, \mathcal{Y}]$ is redundant in S and $\mathcal{Y} \subseteq \mathcal{X}$ then C is redundant in $S|_{\mathcal{X}} \cup \mathcal{X}$.*

Proof. By definition of the c-clause redundancy, there are two cases to consider.

- The first condition leading to redundancy is that $\mathcal{Y}$ is unsatisfiable. In this case, since $\mathcal{Y}$ is a conjunction of literals and $\mathcal{Y} \subseteq \mathcal{X}$, the constraint $\mathcal{X}$ is also unsatisfiable.
- In the second case, there exist n c-clauses $[D_i \,|\, \mathcal{Y}_i] \in S$ $(1 \leq i \leq n)$ such that $\forall i \in [1,n]\, C \succeq D_i$, $\forall i \in [1,n]\, \mathcal{Y}_i \subseteq \mathcal{Y}$ and $\mathcal{Y}', D_1, \dots, D_n \models C$, where $\mathcal{Y}'$ denotes the set of literals in $\mathcal{Y}$ that are lower than C. Since $\mathcal{Y} \subseteq \mathcal{X}$ we deduce that $\forall i \in [1,n]\, \mathcal{Y}_i \subseteq \mathcal{X}$, hence $\forall i \in [1,n]\, D_i \in S|_{\mathcal{X}}$. Since $\mathcal{Y}', D_1, \dots, D_n \models C$, $\mathcal{X}', D_1, \dots, D_n \preceq C$ and $\mathcal{Y}' \cup \{D_1, \dots, D_n\} \subseteq S|_{\mathcal{X}} \cup \mathcal{X}$, C is redundant in $S|_{\mathcal{X}} \cup \mathcal{X}$.

□

Definition 15. *A set of c-clauses S is* saturated *w.r.t. a constraint $\mathcal{X}$ if every c-clause $[C \,|\, \mathcal{Y}]$ such that $\mathcal{Y} \subseteq \mathcal{X}$ that is deducible from S by applying once one of the inference rules is redundant w.r.t. S.*

Theorem 16. *Let $\mathcal{X}$ be a normalized satisfiable constraint. Let S be a set of standard clauses and $S^\star$ be the set obtained from saturating S by applying inference or redundancy deletion rules. If $S^\star$ is saturated w.r.t. $\mathcal{X}$ and $S \models \mathcal{X}^c$, then there exists a constraint $\mathcal{Y} \subseteq \mathcal{X}$ such that $[\Box \,|\, \mathcal{Y}] \in S^\star$.*

Remark 17. *Note that considering only normalized constraints is not restrictive since any constraint is equivalent to a normalized one.*

Proof. Let $S' = S^\star|_{\mathcal{X}} \cup \mathcal{X}$. We first remark that S' is unsatisfiable. Indeed, $S^\star|_{\top} \models S$ since by Proposition 14 all the standard clauses that are removed from S during the saturation process must be redundant in $S^\star|_{\top}$; furthermore, $S^\star|_{\top} \subseteq S^\star|_{\mathcal{X}}$, so that $S' \models S^\star|_{\top} \cup \mathcal{X} \models S \cup \mathcal{X} \models \mathcal{X}^c \cup \mathcal{X}$. We now prove that S' is saturated (in the standard way). We only consider the case where the Superposition rule is applied, the proof for the other rules is similar. Let $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ be two clauses occurring in S', with $l \succ r, u$, and assume that $l \simeq r$ and $l \bowtie u$ are selected in $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ respectively. Let $r \bowtie u \vee P_1 \vee P_2$ be the clause deduced by superposition from the two previous clauses. We distinguish several cases.

- If both $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ occur in $S^\star|_{\mathcal{X}}$, then S contains two c-clauses of the form $[l \simeq r \vee P_1 \mid \mathcal{X}_1]$ and $[l \bowtie u \vee P_2 \mid \mathcal{X}_2]$, where $\mathcal{X}_1, \mathcal{X}_2 \subseteq \mathcal{X}$. It is clear than the Superposition rule applies on these c-clauses, yielding $[r \bowtie u \vee P_1 \vee P_2 \mid \mathcal{X}_1 \wedge \mathcal{X}_2]$. Since $S^\star$ is saturated w.r.t. $\mathcal{X}$, this c-clause is redundant w.r.t. $S^\star$, and since $\mathcal{X}_1 \wedge \mathcal{X}_2 \subseteq \mathcal{X}$ we deduce by Proposition 14 that $r \bowtie u \vee P_1 \vee P_2$ is redundant w.r.t. S'.
- If $l \simeq r \vee P_1$ occurs in $S^\star|_{\mathcal{X}}$ and $l \bowtie u \vee P_2$ occurs in $\mathcal{X}$, then by definition P_2 must be empty, and $S^\star$ contains a c-clause of the form $[l \simeq r \vee P_1 \mid \mathcal{X}_1]$ with $\mathcal{X}_1 \subseteq \mathcal{X}$. Assume that $\bowtie = \not\simeq$. Then the Negative Assertion rule applies on the latter clause, yielding $[r \not\simeq u \vee P_1 \mid \mathcal{X}_1 \wedge l \not\simeq u]$. Since $l \not\simeq u \in \mathcal{X}$, this c-clause must be redundant in S, and Proposition 14 permits to deduce that $r \not\simeq u \vee P_1$ is redundant in $S|_{\mathcal{X}}$. If $\bowtie = \simeq$, then the Positive Assertion rule applies on $[l \simeq r \vee P_1 \mid \mathcal{X}_1]$, yielding $[r \simeq u \vee P_1 \mid \mathcal{X}_1 \wedge l \simeq u]$ and the result follows as in the previous case.
- If $l \simeq r \vee P_1$ occurs in $\mathcal{X}$ and $l \bowtie u \vee P_2$ occurs in $S^\star|_{\mathcal{X}}$, then the proof is similar to the previous case (using only the Positive Assertion rule).
- If both $l \simeq r \vee P_1$ and $l \bowtie u \vee P_2$ occur in $\mathcal{X}$, then $\mathcal{X}$ is not normalized since l occurs at least twice in $\mathcal{X}$, and also occurs as the maximal term of some equation, which contradicts the hypotheses of the theorem.

Since S' is unsatisfiable and saturated, this set necessarily contains $\square$ by completeness of the standard superposition calculus, which entails that $\square \in S^\star|_{\mathcal{X}}$ (since the clauses in $\mathcal{X}$ are unit hence cannot be empty), hence the result. $\square$