A Superposition Strategy for Abductive Reasoning in Ground Equational Logic

Mnacho Echenim, Nicolas Peltier, Sophie Tourret

A Superposition Strategy for Abductive Reasoning in Ground Equational Logic*

Mnacho Echenim, Nicolas Peltier, and Sophie Tourret

University of Grenoble (LIG, Grenoble INP/CNRS)

Abstract. An algorithm is presented for generating implicates of sets of ground, flat, equational clauses. It uses a novel representation of the clause sets that takes the properties of the equality predicate into account in order to ease redundancy elimination. The generation of implicates is performed modulo equivalence and is based on an unordered application of the transitivity axiom with delayed equality tests.

1 Introduction and Motivations

Abduction has been introduced by Peirce [8] as the process of inferring plausible hypotheses from data. This mode of reasoning has many natural applications in computer science. Many verification problems boil down to testing the validity of logical formulæ, possibly modulo some particular theory. The behavior of the system and the expected property are modeled by two logical formulæ H and C and automated reasoning systems are used to verify that the implication $H \Rightarrow C$ holds. In case this formula turns out to be not valid, a counter-example can be constructed [1] to provide a trace of an undesired execution of the system. By analyzing this interpretation, it is often possible to understand where the error comes from and in some cases to correct it. However, this approach is not always satisfactory. First, the counter-example may be very large and complex, with a lot of irrelevant information that can possibly "hide" the origin of the error. Second, it may be too specific, in the sense that it only corresponds to one particular execution of the system and that dismissing this only execution may not be sufficient to fix the system. This leaves to the user the burden of having to infer the general property that can rule out all the undesired behaviors. Consequently, a more useful and informative solution would be to directly infer the missing axioms, or hypotheses, that can be added to H in order to ensure the validity of the formula. Such axioms must be *plausible* and *economical*: for instance the formula C can be added to H, but this is obviously of no use; similarly, explanations that contradict the axioms of the considered theories are obviously irrelevant.

There exists an extensive amount of research on abductive reasoning, mainly in propositional logic, with numerous applications in, e.g., artificial intelligence. Abduction can be performed in a top-down manner, by allowing some hypotheses

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

to be asserted instead of being proven. However it is more often reduced to a consequence-generation problem: indeed, by contrapositive, the implication $H \Rightarrow C$ holds iff $\neg H$ is a logical consequence of $\neg C$. Thus explanations of C can be generated from the derivation of the *implicates*[1] of $\neg C$. In general, these explanations are further restricted to ensure relevance: only explanations defined on a particular set of symbols, called the *abducible symbols* are considered.

Most automated theorem provers focus on proving that one particular property holds (usually by contradiction), and thus they are not well-adapted for the "blind" generation of implicates. Although saturation-based provers work by generating consequences of sets of axioms, the application of the inference rules is strongly restricted (for obvious efficiency reasons), which prevents the generation of some potentially relevant implicates. Existing approaches for computing implicates use either variants of the (unordered) resolution rule, together with specific redundancy criteria and strategies ensuring efficiency [11, 6, 5, 2], or decomposition-based approaches in the spirit of the DPLL method, which compute implicates by recursively decomposing them into smaller pieces [10, 9, 7]. In [3], we devised a variant of the *superposition calculus* that is specifically tuned for generating ground consequences of sets of axioms. Given a set of first-order clauses S, this calculus is able to generate a set of ground, flat, clauses $T_\infty(S)$ that describes all possible consequences of S that are of interest: if C is an implicate of S built over abducible symbols, then it is also an implicate of $T_\infty(S)$. Thus, the formula $\neg T_\infty(S)$ can be viewed as a representation of the set of all plausible hypotheses ensuring the unsatisfiability of S. However, returning the set $T_\infty(S)$ to a user is not fully satisfactory, since this set may be very large and contain a lot of redundant information. In practice, it would be more useful to return only the most general clauses that are logical consequences of $T_\infty(S)$, which correspond to the most economical explanations of S. Such clauses are the *prime implicates* of $T_\infty(S)$.

2 Generating Prime Implicates

A first method to automatically generate prime implicates built over abducible symbols is given in [3]. It applies the resolution rule on the clause set $T_\infty(S)$ enriched with all ground instances of the equality axioms on the considered signature. It is shown that this approach is complete, in the sense that every prime implicate is eventually obtained. However, it is also very inefficient, because the systematic instantiation of the equality axioms produces numerous clauses, with a huge number of potential resolvents. In particular, the resolution rule will derive clauses containing constant symbols not occurring in the premises in $T_\infty(S)$: for instance a clause $a \not\simeq b$, will produce, together with the transitivity axiom $a \not\simeq c \vee c \not\simeq b \vee a \simeq b$, one clause $a \not\simeq c \vee c \not\simeq b$ for each constant symbol c in the signature, even if there is no logical connection between c and a or b. From a practical point of view, such inferences are rather cumbersome, and it would be obviously better to have a calculus in which the properties of the equality predicate are directly encoded as built-in features of the inference process, rather

[1] A formula F' is an implicate of a formula F iff F' is not a tautology and $F \models F'$.

than having to express them explicitly as axioms. A natural candidate is the superposition calculus, but it is easy to see that it is too restrictive for our purpose: for example, a prime implicate of the clauses $a \not\simeq b$ and $c \simeq d$, is $c \not\simeq a \vee d \not\simeq b$, yet there is no way of generating this clause from the two premises by superposition, even if ordering conditions are relaxed. It is thus natural to search for new superposition-based strategies for generating implicates. On one hand, such strategies must be more relaxed than the standard superposition inference rules, so that all prime implicates can be obtained, but on the other hand, they should be more focused than the blind application of the resolution rule, in order to yield a more efficient and hopefully scalable algorithm. We devise such a strategy in the present paper.

3 Representation of Clauses Modulo Equality

In propositional logic, detecting redundant clauses is an easy task, since a (non-tautological) clause C is a logical consequence of D iff D is a subclause of C. Thus C is redundant in a clause set iff there exists a clause $D \in S$ such that $D \subseteq C$. The literals can be totally ordered to handle commutativity and the clause set S can be represented as a trie (a tree-based data-structure commonly used to represent strings [4]), so that inclusion can be tested efficiently. However, in ground equational logic, the above property does not hold anymore: for instance the clause $a \not\simeq b \vee b \simeq c$ is a logical consequence of $a \simeq c$ but obviously $a \simeq c$ is not a subclause of $a \not\simeq b \vee b \simeq c$. Thus testing clause inclusion is no longer sufficient and representing clause sets as tries would yield many undesired redundancies: for instance the clauses $a \not\simeq b \vee b \simeq c$ and $a \not\simeq b \vee a \simeq c$ would be both stored, although they are equivalent. Our first task is thus to devise a new redundancy criterion that generalizes subsumption, together with a new way of representing clauses, that takes into account the special properties of the equality predicate. To this purpose we show how to normalize ground clauses according to a total ordering on constant symbols, and we introduce a new notion of *projection*.

Let $\prec$ be a fixed, total ordering on constant symbols and C be a clause. The *C-representatives* of a constant a, a literal[2] l and a clause D are respectively defined as follows: $a_{\downarrow C} = \min_{\prec}\{b \mid b \not\simeq a \models C\}$; $l_{\downarrow C} = a_{\downarrow C} \bowtie b_{\downarrow C}$ when $l = a \bowtie b$; and $D_{\downarrow C} = \{l_{\downarrow C} \mid l \in D\}$. The literal $l_{\downarrow C}$ (resp. the clause $D_{\downarrow C}$) is called the *projection of l (resp. D) on C*. It can be verified that every clause C is equivalent to the clause: $C_{\downarrow} \stackrel{\text{def}}{=} \bigvee_{a \in C, a \neq a_{\downarrow C}} a \not\simeq a_{\downarrow C} \vee \bigvee_{a \simeq b \in C} a_{\downarrow C} \simeq b_{\downarrow C}$. The clause $C_{\downarrow}$ is the *normal form* of C and it is unique.

Clauses in normal form are stored in a tree data-structure, called a *clausal tree*, specifically tailored to store sets of literals while taking into account the usual properties of the equality predicate. As in tries, the edges of the tree are labeled by literals and the leaves are labeled either by $\Box$ (representing the empty clause) or by $\emptyset$ (failure node). The set of clauses $\mathcal{C}(T)$ represented by a tree T is the disjunction of literals labeling a path from the root to those leaves labeled by $\Box$. Failure nodes are useful mainly to represent empty sets – in fact they can always be eliminated by straightforward simplification rules, except

[2] When we write $l = a \bowtie b$, the symbol $\bowtie$ stands for $\simeq$ or $\not\simeq$.

if the root itself is labeled by $\emptyset$. A few constraints are imposed on the literals labeling the edges of a clausal tree – the main one being that negative literals occur before positive ones in all branches of the tree – in order to guarantee that the clauses represented by a clausal tree are in normal form, and that this data-structure can be used to test redundancy efficiently. The conditions permit to design an efficient test whether a given clause is redundant w.r.t. those stored in the tree (forward subsumption) and conversely to delete from the tree all clauses that are redundant w.r.t. a newly generated clause (backward subsumption).

Theorem 1. *Let C and D be two clauses. Assume that C is not a tautology. $D \models C$ iff every negative literal in $D_{\downarrow C}$ is a contradiction and every positive literal in $D_{\downarrow C}$ is also in $C_{\downarrow C}$.*

We will use a slightly more restrictive version of this criterion for redundancy elimination, by viewing the clauses as multisets and by imposing that the positive literals in $D_{\downarrow C}$ are mapped to pairwise distinct literals in $C_{\downarrow C}$. This additional restriction is necessary to prevent the factors of a clause from being redundant w.r.t. the initial clause. For instance, the clause $a \simeq b \vee a \not\simeq a'$ will not be redundant w.r.t. $a \simeq b \vee a' \not\simeq b \vee a \not\simeq a'$, although $a \simeq b \vee a' \not\simeq b \vee a \not\simeq a' \models a \simeq b \vee a \not\simeq a'$.

Algorithm 1 isEntailed(C, T)

Require: C is a clause in normal form and T is an $\mathcal{RN}$-clausal tree.
Ensure: isEntailed$(C, T) = \mathit{true} \Leftrightarrow \exists D \in \mathcal{C}(T),\ D \models C$

```
if T = □ then
    return true
end if
if C = □ then
    return false
end if
l_1 ← min_< {l ∈ C}
for all (l, T') ∈ T such that l ≥ l_1 do
    if l_1 = a ≯ b, with a ≻ b then
        if l = l_1 then
            if isEntailed(C\l_1, T') then
                return true
            end if
        else if ¬(l = a ≯ c), with a ≻ c then
            if isEntailed(C\l_1, (T'|_l^↑)[a := b]) then
                return true
            end if
        end if
    else if l ∈ C then
        if isEntailed(C\l, T') then
            return true
        end if
    end if
end for
return false
```

The first algorithm is invoked on a clause C and a tree T, and returns *true* if and only if there exists a clause D in $\mathcal{C}(T)$ such that C is redundant w.r.t. D. To test this entailment, the algorithm performs a depth-first traversal of T and attempts to project every encountered literal on C. If a literal cannot be projected, the exploration of the subtree associated to this literal is useless, so the algorithm switches to the following literal. As soon as a clause entailing C is found, the traversal halts and *true* is returned.

The second algorithm deletes from a tree T all clauses entailed by C. It performs a depth-first traversal of T and attempts to project C on every clause in $\mathcal{C}(T)$, deleting those on which such a projection succeeds. As soon as a projection is identified as impossible, the exploration of the associated subtree halts and the algorithm moves on to the next clause. When every literal in C has been projected, all the clauses represented in the current subtree are entailed by C, and are therefore deleted.

Algorithm 2 PRUNEENTAILED(C, T)

Require: C is a clause in relaxed normal form, T is an $\mathcal{N}$-clausal tree and ISENTAILED$(C, T) = \mathit{false}$
Ensure: $\forall D \in \mathcal{C}(T)$, $C \not\models D$
  if $C = \square$ **then**
    $T \leftarrow \emptyset$
    exit
  end if
  if $T = \square$ **then**
    exit
  end if
  $l_1 \leftarrow \min_{<} \{l_i \in C\}$
  for all $(l, T') \in T$ such that $l \leq l_1$ **do**
    if $l_1 = l$ **then**
      PRUNEENTAILED$(C \backslash l_1, T')$
    else
      if $l = a \simeq b$ **then**
        PRUNEENTAILED(C, T')
      else if $l = a \not\simeq b$, with $a \succ b$ **and** $\nexists c$, $l_1 = a \not\simeq c$, with $a \succ c$ **then**
        PRUNEENTAILED$(C[a := b], T')$
      end if
    end if
  end for

4 Generation of Implicates

The clausal trees described above are used during the generation of implicates to store the non-redundant clauses generated by the following inference rules:

$$\frac{a \simeq b \vee C \qquad a' \simeq c \vee D}{a \not\simeq a' \vee b \simeq c \vee C \vee D} \qquad \frac{a \simeq b \vee a' \simeq b' \vee C}{a \simeq b \vee a \not\simeq a' \vee b \not\simeq b' \vee C}$$

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

The premises are assumed to be in normal form and the conclusion is normalized before being stored. The first rule is similar to the usual paramodulation rule, except that the unification between the terms a and a' is omitted and replaced by the addition of the literal $a \not\simeq a'$ ensuring that these terms are semantically equal. Similarly, the second rule factorizes the literals $a \simeq b$ and $a' \simeq b'$ and adds the conditions $a \not\simeq a'$ and $b \not\simeq b'$. The last rule corresponds to an application of the factorization rule on the negative literals $a_i \not\simeq b_i$, followed by a paramodulation step which removes these literals, while adding the conditions ensuring that $a_i = c$ and $b_i = d$. Note that no ordering condition is imposed. It is easy to see that the addition of ordering restrictions (as in the usual version of the rules) makes the calculus incomplete. For instance, assume that $S = \{a \not\simeq b \vee c \simeq d, a \simeq b\}$, with $a \prec b \prec c \prec d$. Then $S \models c \simeq d$,

but $a \not\simeq b$ is not maximal in $a \not\simeq b \vee c \simeq d$, thus it is impossible to infer the clause $c \simeq d$ if paramodulation is restricted to maximal literals. Similarly, if $S' = \{a \not\simeq b, c \simeq b\}$, then $S' \models a \not\simeq c$, but $a \not\simeq c$ can only be derived if the paramodulation rule is applied on the non-maximal term a in $a \not\simeq b$ (replacing a by b in $a \not\simeq b$ yields a contradiction and the condition $a \not\simeq c$ is added in the clause) or from the non-maximal term b in $c \simeq b$ (replacing b by c in $a \not\simeq b$ yields $a \not\simeq c$).

Some of the literals in the conclusions, more precisely the conditions $a \not\simeq a'$, $b \not\simeq b'$, $a_i \not\simeq c$ and $d \not\simeq b_i$ can be "frozen" in the sense that no further inference is allowed within them (these literals will eventually remain – after normalization – in the considered prime implicate). Although this remark can dismiss many inferences, its practical interest remains unclear, since the frozen literals have to be considered apart when applying the redundancy detection, which may prevent the removal of numerous clauses.

A set of clauses S is *saturated* iff for every clause C that can be derived from S using these rules, there exists a clause $C' \in S$ such that C is redundant w.r.t. C'. The following theorem states the completeness of this approach:

Theorem 2. *Let S be a set of ground flat clauses in normal form. If S is saturated and $S \models C$ then there exists a clause $C' \in S$ such that $C' \models C$.*

A seemingly natural idea would be to restrict the last rule to the case $n = 1$ (i.e. to remove the preliminary implicit factorization step when paramodulating into negative literals). However, this makes the calculus incomplete. For instance, consider the set: $S = \{c \not\simeq a \vee d \not\simeq a, a \simeq b \vee c \not\simeq b \vee d \not\simeq b\}$. It is clear that $S \models c \not\simeq b \vee d \not\simeq b$ and that S is saturated if the negative paramodulation rule is applied on a unique literal only.

5 Conclusion

We have devised a calculus for generating prime implicates of clause sets defined over equations and disequations between constants. Such an algorithm is much more restricted and thus (probably) more efficient than the naive approach consisting in applying the resolution calculus on equality axioms. In particular, the clauses are taken modulo equivalence, and efficient data-structures are used to represent clause sets. Algorithms are provided for updating such data-structures and detecting redundancy. The generation of the implicates is performed by applying a constrained paramodulation rule, where equations permitting the application of the transitivity axiom are allowed to be asserted instead of being proved. While such a rule is far more restricted than the resolution rule, its definition probably leaves room for improvements: in particular, no ordering restriction is considered in the current version. This is to some extent unavoidable if one wants to derive all implicates[3], but, still, some partial ordering conditions can probably be enforced while retaining completeness, and we are currently

[3] For instance, given the clauses $a \simeq b \vee c \not\simeq d$ and $c \simeq d$, one should be able to prove that $a \simeq b$ is an implicate, hence to perform inferences on $c \not\simeq d$ even if $a, b \succ c, d$.

investigating this problem, as well as working on an implementation of the calculus.

References

1. R. Caferra, A. Leitsch, and N. Peltier. *Automated Model Building*, volume 31 of *Applied Logic Series*. Kluwer Academic Publishers, 2004.
2. 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.
3. M. Echenim and N. Peltier. A Calculus for Generating Ground Explanations. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'12)*. Springer LNCS, 2012.
4. E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.
5. P. Jackson. Computing prime implicates incrementally. *Automated Deduction CADE-11*, pages 253–267, 1992.
6. A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2):185–206, 1990.
7. A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 250–264, 2009.
8. C. S. Peirce. *Philosophical Writings of Peirce*. Dover Books, Justus Buchler editor, 1955.
9. A. Ramesh, G. Becker, and N. Murray. Cnf and dnf considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18(3):337–356, 1997.
10. R. Rymon. An se-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, 11(1):351–365, 1994.
11. P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *Electronic Computers, IEEE Transactions on*, (4):446–456, 1967.