Identifying non-redundant literals in clauses with uniqueness propagation

Hendrik Blockeel

Department of Computer Science, KU Leuven

Abstract. Several authors have proposed increasingly efficient methods for computing the least general generalization of two clauses under theta-subsumption. The most expensive step in this computation is the reduction of the clause to its minimal size. In general, this requires testing all literals for redundancy, which in the worst case requires a theta-subsumption test for each of them. In this paper, we present a small result that is complementary to earlier published results. It basically extends the conditions under which a literal can be decided non-redundant without running a theta-subsumption test. Preliminary experiments show a speedup factor of about 3 on one dataset when this new result is used.

1 Introduction

Several ILP systems rely on computing the least general generalization (lgg) of two clauses under theta-subsumption. This computation consists of two steps. In the first step, a clause is constructed that contains all pairwise lgg's of literals from the first and second clause. This clause may contain many redundant literals, therefore, in the second step, which is called *reduction*, all redundant literals are removed from this clause. Checking whether a literal is redundant boils down to a theta-subsumption test in the worst case. Therefore, while the first step has polynomial complexity ($O(mn)$, with m and n the number of literals in the first and second clause, respectively), the second step is NP-hard ($O(n^m)$). Unsurprisingly, attempts to make the computation of the lgg more efficient focus on reducing the number of theta-subsumption tests needed (in addition to using an efficient theta-subsumption test, which is a separate problem).

In this paper, we present a result that, while simple, does not seem to have made it into the literature. We show that exploiting this result can substantially reduce the number of theta-subsumption tests needed during the reduction.

2 Theta-subsumption, lgg and reductions

We assume familiarity with standard terminology from logic programming, and with the Prolog programming language.

2.1 Definitions and notation

Definition 1 (theta-subsumption). *A clause c theta-subsumes another clause d, denoted $c \preceq_\theta d$, if and only if there exists a variable substitution θ such that $c\theta \subseteq d$.*

Definition 2 (lgg). *The least general generalization under theta-subsumption (briefly, lgg) of two clauses c and d is e if and only if $e \preceq_\theta c$, $e \preceq_\theta d$, and there does not exist a clause $e' \neq e$ such that $e \preceq_\theta e'$, $e' \preceq_\theta c$ and $e' \preceq_\theta d$.*

Theta-subsumption is a semi-order; it is reflexive and transitive but not anti-symmetric. Different clauses may subsume each other; they are called variants of each other.

Definition 3 (variants). *Two clauses c and d are variants, denoted $c \sim d$, if $c \preceq_\theta d$ and $d \preceq_\theta c$.*

Variants are logically equivalent to each other.

Definition 4 (redundant literals). *A literal l in a clause c is redundant if and only if $c \setminus \{l\} \sim c$.*

Let $|c|$ denote the number of literals in c.

Definition 5 (reduced clauses). *A clause c is reduced if there is no $c' \sim c$ such that $|c'| < |c|$.*

In other words, a clause is reduced if there is no shorter clause equivalent to it. Reducing a clause c means computing its shortest variant.

2.2 Computing the lgg

The lgg of two clauses can be computed using an algorithm proposed by Plotkin [5]. In a first phase, the algorithm constructs the lgg of two clauses c and d by including each literal that is a least generalization of a literal in c and one in d. The resulting clause can contain many redundant literals. In a second phase, it is *reduced*: all redundant literals are removed.

Gottlob and Fermüller (1993) proposed the following algorithm for reduction of clauses:

```
for each literal $l$ in the original clause $c$:
    if $c \preceq_\theta c \setminus \{l\}$ then remove $l$ from $c$
```

This algorithm takes $|c|$ calls to theta-subsumption. As indicated by Gottlob and Fermüller, this number can be reduced, by skipping those literals for which it is obvious that they cannot be removed. More generally, to reduce a clause more efficiently, the following scheme can be used:

1. efficiently identify non-redundant literals
2. efficiently identify redundant literals

3. for the remaining literals, determine redundancy by means of a subsumption test

Gottlob and Fermüller point out that the test $c \preceq_\theta c \setminus \{l\}$ can only succeed if there exist a θ such that $l\theta \in c \setminus \{l\}$, in other words, there must be another literal $l' \in c$, besides l, such that $l\theta = l'$. A literal l that has no matching literal l' is certainly non-redundant.

Malobert and Suzuki point out that, when $c\theta \subseteq c \setminus \{l\}$, all literals in $c \setminus \{l\} \setminus c\theta$ can be concluded to be redundant, not just l. They provide a method for computing the θ that minimizes $|c\theta|$, thus identifying a maximum number of redundant literals using one complete subsumption test (a complete subsumption test is one that returns $\{\theta | c\theta \subseteq d\}$ rather than just a boolean that indicates the non-emptiness of this set).

The main contribution of this paper is that we generalize the conditions under which literals can easily be identified as non-redundant, thus increasing the impact of (1).

3 Efficiently identifying non-redundant literals

We now generalize the conditions under which a literal l can safely be assumed non-redundant.

Definition 6. *Given a clause c, a literal $l' \in c$ matches a literal $l \in c$ if $\exists\theta : l\theta = l'$.*

Definition 7. *A literal l in a clause c is called unique if and only if there is no literal $l' \in c \setminus \{l\}$ that matches l.*

Now consider the following procedure:

```
function ReduceUP(clause C):
    U := $\emptyset$
    repeat
        for all $l \in C$:
            if there is no $l'$ in $C \cup U \setminus \{l\}$ that matches $l$
            then remove $l$ from $C$, add $l$ to $U$
        skolemize $U$
    until C does not change anymore
    for all $l \in C$: if $U \cup C \preceq_\theta U \cup C \setminus \{l\}$ then remove $l$ from $C$
    deskolemize $U \cup C$
    return $U \cup C$
```

The for-all loop in this algorithm moves all unique literals from C to U. Literals in U are certainly non-redundant and need not be tested afterwards.

After this for-all loop comes a skolemization step. Skolemization instantiates each variable in U with a different and new constant (that is, one that does not yet occur in $U \cup C$). Note that U and C share variables; if a variable in

U is instantiated, its occurrences in C are too. This may introduce new unique constants in C, and therefore, a literal that had a matching literal earlier on may no longer have one now. A new check for non-matched literals is then run. More literals may be added to U as a result; more variables will be instantiated in the new skolemization step, and so on, until C no longer changes. At that point, the literals in C are tested for redundancy using theta-subsumption.

Example 1. Consider $C = \{p(a, Y), p(Y, b), p(X, b), p(X, Z)\}$.

In the first run, $p(a, Y)$ is the only literal that does not match any other literals. Therefore, after one execution of the outer loop, $U = \{p(a, Y)\}$ and $C = \{p(Y, b), p(X, b), p(X, Z)\}$. We now skolemize U, which results in $U = \{p(a, y)\}$ and $C = \{p(y, b), p(X, y), p(X, Z)\}$ (where y is the constant that variable Y was instantiated to). At this point, $p(y, b)$ no longer matches any other literal and therefore joins U, giving $U = \{p(a, y), p(y, b)\}$ and $C = \{p(X, b), p(X, Z)\}$. At this point U contains no variables, therefore skolemization does not change anything and C will not change. The algorithm will now only test the literals in C for redundancy using theta-subsumption. Both literals in C turn out to be redundant (by $\theta = \{X/y, Z/b\}$), yielding as final result $U = \{p(a, y), p(y, b)\}$ and $C = \emptyset$, and after deskolemization, returning $\{p(a, Y), p(Y, b)\}$ as the reduced clause.

It is intuitively clear that this procedure is sound: if a literal l in C becomes unique through the skolemization (while it wasn't before), this is because it shared a variable with a literal u in U. Hence, any substitution θ that could be used to make l equal to another literal l' (which is necessary to have $(U \cup C)\theta \subseteq U \cup C \setminus \{l\}$) would have an effect on u as well, and would map u to a literal outside $U \cup C$ (since no substitution exists that maps u to a literal in $U \cup C$, by definition of uniqueness of u).

The final subsumption tests in our algorithm are performed using the partially skolemized clause. Indeed, it makes no difference whether these tests are run on the deskolemized or partially skolemized clause: by definition of uniqueness, we know that the literals in u cannot map onto any other literal than themselves, therefore the variables in them cannot occur in any substitution except one that assigns them to themselves, therefore replacing these variables by constants does not make a difference for the outcome of the test. The subsumption test may be faster when performed on the partially skolemized clause, though, simply because this clause contains fewer variables (and the number of variables affects the efficiency of theta-subsumption).

4 Discussion

The algorithm presented above causes a kind of "propagation of uniqueness" (hence its name, ReduceUP). While it trivial that a literal with a unique predicate symbol or a unique constant must be unique (hence, non-redundant), the skolemization of unique literals can cause their neighbors (that is, the literals that share a variable with them) to become unique too. This may substantially increase the number of literals considered unique and therefore trivially non-redundant.

The above result is complementary to existing results on lgg computation. Both the procedure proposed by Gottlob and Fermüller [3] and the one by Maloberti and Suzuki [4] can be made faster if more literals can be decided beforehand to be non-redundant.

5 Experiments

We performed preliminary experiments on a dataset used by Becerra-Bonache et al. [1, 2] for language learning experiments. Processing the first 100 examples gave rise to 525 lgg computations. The lggs contained on average 58 literals. Of these, 5.13 on average were unique in the sense of Gottlob, and 11.26 were unique after " uniqueness propagation".

Our lgg implementation uses another, more trivial, optimization: literals that do not share any variables with other literals and subsume a unique literal are automatically removed (indeed, as they do not share variables with any other literals, their matching substitution cannot be incompatible with substitutions for other literals). This further reduces the number of remaining literals that needs to be tested by subsumption. With an overall CPU time of 7.1s for processing 100 examples, the version with uniqueness propagation is over 3 times faster than the one with standard uniqueness (25.0 seconds). This factor 3 is higher than the reduction of the number of subsumption tests, but as said before, the tests themselves tend to become more efficient too.

6 Conclusions

The computation of the lgg of two clauses contains a reduction step, which itself is NP-hard and generally has to rely on multiple theta-subsumption tests. The fewer such tests are needed, the more efficient the lgg computation becomes. "Unique" literals, which have no other literals that match it, can trivially be excluded from such testing. In this paper, we have showed that the concept of uniqueness can be broadened, so that more literals fulfill the criterion, and that this can lead to faster computation of the lgg.

This work is preliminary. A more extensive experimental comparison of lgg computation methods would be useful; this comparison should be done on a wider range of datasets, and should assess not only the effect of uniqueness propagation in a simple implementation of lgg, but also when combined with a more advanced version such as Maloberti and Suzuki's Jivaro.

References

1. Becerra-Bonache, L., Blockeel, H., Galván, M., Jacquenet, F.: A first-order-logic based model for grounded language learning. In: Advances in Intelligent Data Analysis XIV - 14th International Symposium, IDA 2015, Saint Etienne, France, October 22-24, 2015, Proceedings. pp. 49–60 (2015)

2. Becerra-Bonache, L., Blockeel, H., Galván, M., Jacquenet, F.: Relational grounded language learning. In: Proceedings of the 22nd European Conference on Artificial Intelligence (2016), to appear
3. Gottlob, G., Fermüller, C.G.: Removing redundancy from a clause. Artif. Intell. 61(2), 263–289 (1993)
4. Maloberti, J., Suzuki, E.: An efficient algorithm for reducing clauses based on constraint satisfaction techniques. In: Inductive Logic Programming, 14th International Conference, ILP 2004, Porto, Portugal, September 6-8, 2004, Proceedings. pp. 234–251 (2004)
5. Plotkin, G.D.: Machine Intelligence 5, chap. A note on inductive generalization, pp. 153–163. Edinburgh University Press (1970)