θ-Subsumption and Resolution: A New Algorithm

S. Ferilli, N. Di Mauro, T.M.A. Basile, and F. Esposito

Dipartimento di Informatica
Università di Bari
via E. Orabona, 4 – 70125 Bari – Italia
{ferilli,nicodimauro,basile,esposito}@di.uniba.it

Abstract. Efficiency of the first-order logic proof procedure is a major issue when deduction systems are to be used in real environments, both on their own and as a component of larger systems (e.g., learning systems). This paper proposes a new θ-subsumption algorithm that is able to return the set of all substitutions by which such a relation holds between two clauses without performing backtracking. Differently from others proposed in the literature, it can be extended to perform resolution, also in theories containing recursive clauses.

1 Introduction

Logic Programming [6] is a computer programming approach based on the representation of programs as first-order logic theories made up of Horn clauses, whose execution is reduced to proving statements in the given theory. Since the classical provability relation, logic implication, is undecidable [10], the weaker but decidable generality relation of θ-subsumption is often used in practice. Given C and D clauses, C θ-subsumes D (often written $C \leq D$) iff there is a substitution θ such that $C\theta \subseteq D$. A substitution is a mapping from variables to terms, often denoted by $\theta = \{X_1 \rightarrow t_1, \ldots, X_n \rightarrow t_n\}$, whose application to a clause C, denoted by $C\theta$, rewrites all the occurrences of variables X_i $(i = 1 \ldots n)$ in C by the corresponding term t_i.

Since in this framework program execution corresponds to proving a theorem, efficiency of the generality relation used is a key issue that deserves great attention, whichever application Logic Programming is used for. In Theorem Provers, explosion of the possible interactions between clauses is often limited by deleting all clauses that are discovered to be already subsumed by other clauses in the theory. In Inductive Logic Programming (ILP), a large amount of tests is needed to check completeness and consistency of new hypotheses against all given examples. Another exploitation of θ-subsumption tests is to compute the reduction of clauses, i.e. a clause that is equivalent to a given one but from which all redundant (superfluous) literals have been deleted.

In the following, we will assume that C and D are Horn clauses having the same predicate in their head, and that the aim is checking whether C θ-subsumes D. Note that D can always be considered ground (i.e., variable-free) without loss

N. Zhong et al. (Eds.): ISMIS 2003, LNAI 2871, pp. 384–391, 2003.

of generality. Indeed, in case it is not, each of its variables can be replaced by a new constant not appearing in C nor in D, obtaining a new clause D', and it can be proven that C θ-subsumes D iff C θ-subsumes D'. Since the test if C θ-subsumes D' can be cast as a refutation of $\{C\} \cup \neg D'$, a basic algorithm can be obtained in Prolog by asserting C plus all the literals in the body of D', and finally querying the head of D'. The outcome is computed by Prolog through SLD resolution [8], which can be very inefficient under some conditions.

Example 1. Given the following two clauses:

`h(`X`) :- p(`X`,`X_1`),p(`X`,`X_2`), ... , p(`X`,`X_n`),q(`X_n`).`
`h(`c`) :- p(`c`,`c_1`),p(`c`,`c_2`), ... ,p(`c`,`c_m`).`

SLD-resolution will have to try all m^n possible mappings (backtrackings) before realizing that the former does not subsume the latter because of the lack of property q. Thus, the greater n and m, the sooner it will not be able to compute subsumption between them two clauses within acceptable time.

The next section presents related work in this field; then, Section 3 introduces the new subsumption algorithm, while Section 4 shows how it can be used to perform resolution. Lastly, Section 5 concludes the paper.

2 Related Work

The great importance of finding efficient θ-subsumption algorithms is reflected by the amount of work carried out so far in this direction in the literature. Two classical algorithms are based on resolution: The former, by Chang and Lee [1], carries out a complete resolution of each literal in C with all possible literals in the negation of D, whereas the latter, by Stillman [11], chooses one at each step and then exploits backtracking in case of wrong choice. The latter uses backtracking to avoid the computation of further, useless unifications once a solution is found. Based on such considerations, Gottlob and Leitsch [4] defined a new backtracking algorithm that attacks the problem complexity by first partitioning the clause into independent subsets, and then applying resolution separately to each of them, additionally exploiting a heuristic that resolves each time the literal with the highest number of variables that occur also in other literals.

A more formal approach was then taken by Kiets and Lübbe [5]. They first identify the subset of C that deterministically θ-subsumes D, and then separate the rest of C into independent parts that can be handled separately by θ-subsumption algorithms. Scheffer, Herbrich and Wysotzki [9] transposed the problem into a graph framework, in which additional techniques can be exploited. They take into account not just single literals, but also their 'context' (i.e., the literals to which they are connected via common variables). Indeed, by requiring that two literals have the same context in order to be matched, the number of literals in C that have a unique matching candidate in D potentially grows. The remaining (non-determinate) part of C is then handled by mapping the subsumption problem onto a search for the maximum clique in a graph, for which known efficient algorithms are known, and can be properly tailored.

All the techniques presented so far rely on backtracking, and try to limit its effect by properly choosing the candidates in each tentative step. Hence, all of them return only the first subsuming substitution found, even if many exist.

Finally, Maloberti and Sebag [7] face the problem of θ-subsumption by means of a Constraint Satisfaction Problem (CSP) approach by transforming each literal involved in the hypothesis into a CSP variable with proper constraints encoding the θ-subsumption structure. Given such a representation, different versions of a correct and complete θ-subsumption algorithm, named *Django*, were built, each implementing different (combinations of) CSP heuristics. Experiments in various domains prove a difference in performance of several orders of magnitude in favor of Django compared to the algorithms described above, thus comparison of new algorithms to just this system should be enough to evaluate their efficiency. Note that Django only gives a binary (*yes* or *no*) answer to the subsumption test, without providing any matching substitution in case of positive outcome.

3 A New Matching Algorithm

Ideas presented in Section 2 aim at identifying subparts of the given clauses for which the θ-subsumption test can be computed with reduced complexity, and base their efficiency on the retrieval of just one solution (substitution). Then, only classical algorithms can be applied to the remaining parts. In those cases, the CSP approach proves very efficient, but at the cost of not returning any substitution by which the matching holds. This suggests that they would not be able to support resolution, in particular when recursive clauses are involved. The proposed algorithm, on the contrary, returns *all* possible matching substitutions, without performing any backtracking in their computation. Such a feature is important, since the found substitutions can be made available to further matching problems, thus allowing to perform resolution.

Before discussing the new procedure proposed in this paper, it is necessary to give some definitions on which the algorithm is based. In the following we will assume C and D to be clauses, such that C is constant-free and D is ground.

Definition 1 (Matching substitution). *A* matching substitution *from a literal l_1 to a literal l_2 is a substitution μ, such that $l_1\mu = l_2$.*

The set of all matching substitutions from a literal $l \in C$ to some literal in D is denoted by [2]:

$$uni(C, l, D) = \{\mu \mid l \in C, l\mu \in D\}.$$

Let us start by defining a structure to compactly represent sets of substitutions.

Definition 2 (Multisubstitutions). *A* multibind *is denoted by $X \to T$, where X is a variable and $T \neq \emptyset$ is a set of constants. A* multisubstitution *is a set of multibinds $\Theta = \{X_1 \to T_1, \ldots, X_n \to T_n\} \neq \emptyset$, where $\forall i \neq j : \ X_i \neq X_j$.*

Informally, a *multibind* identifies a set of constants that can be associated to a variable, while a *multisubstitution* represents in a compact way a set of possible substitutions for a tuple of variables. In particular, a single substitution is represented by a multisubstitution in which each constants set is a singleton.

Example 2. $\Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$ is a multisubstitution. It contains 3 multibinds, namely: $X \to \{1,3,4\}$, $Y \to \{7\}$ and $Z \to \{2,9\}$.

Given a multisubstitution, the set of all substitutions it represents can be obtained by choosing in all possible ways one constant for each variable among those in the corresponding multibind.

Example 3. The set of all substitutions represented by the multisubstitution $\Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$ is the following:
$\{\{X \to 1, Y \to 7, Z \to 2\}, \{X \to 1, Y \to 7, Z \to 9\}, \{X \to 3, Y \to 7, Z \to 2\}, \{X \to 3, Y \to 7, Z \to 9\}, \{X \to 4, Y \to 7, Z \to 2\}, \{X \to 4, Y \to 7, Z \to 9\}\}$.

Definition 3 (Union of multisubstitutions). *The union of two multisubstitutions* $\Theta' = \{\overline{X} \to T', X_1 \to T_1, \ldots, X_n \to T_n\}$ *and* $\Theta'' = \{\overline{X} \to T'', X_1 \to T_1, \ldots, X_n \to T_n\}$ *is the multisubstitution defined as*

$$\Theta' \sqcup \Theta'' = \{\overline{X} \to T' \cup T''\} \cup \{X_i \to T_i\}_{1 \le i \le n}$$

Note that the two input multisubstitutions must be defined on the same set of variables and must differ in at most one multibind.

Example 4. The union of two multisubstitutions
$\Sigma = \{X \to \{1,3\}, Y \to \{7\}, Z \to \{2,9\}\}$ and $\Theta = \{X \to \{1,4\}, Y \to \{7\}, Z \to \{2,9\}\}$, is: $\Sigma \sqcup \Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$ (the only different multibinds being those referring to variable X).

Definition 4 (merge). *Given a set* $\mathcal{S}$ *of substitutions on the same variables,* $\mathrm{merge}(\mathcal{S})$ *is the set of multisubstitutions obtained according to Algorithm 1.*
Example 5. $\mathrm{merge}(\{\{X \to 1, Y \to 2, Z \to 3\}, \{X \to 1, Y \to 2, Z \to 4\}, (X \to 1, Y \to 2, Z \to 5\}\}) = \mathrm{merge}(\{\{X \to \{1\}, Y \to \{2\}, Z \to \{3,4\}\}, \{X \to \{1\}, Y \to \{2\}, Z \to \{5\}\}\}) = \{\{X \to \{1\}, Y \to \{2\}, Z \to \{3,4,5\}\}\}$.
This way we can represent 3 substitutions with only one multisubstitution.

Definition 5 (Intersection of multisubstitutions). *The intersection of two multisubstitutions* $\Sigma = \{X_1 \to S_1, \ldots, X_n \to S_n, Y_1 \to S_{n+1}, \ldots, Y_m \to S_{n+m}\}$ *and* $\Theta = \{X_1 \to T_1, \ldots, X_n \to T_n, Z_1 \to T_{n+1}, \ldots, Z_l \to T_{n+l}\}$, *where* $n, m, l \ge 0$ *and* $\forall j,k : Y_j \neq Z_k$, *is the multisubstitution defined as:*

$$\Sigma \sqcap \Theta = \{X_i \to S_i \cap T_i\}_{i=1\ldots n} \cup \{Y_j \to S_{n+j}\}_{j=1\ldots m} \cup \{Z_k \to T_{n+k}\}_{k=1\ldots l}$$

iff $\forall i = 1 \ldots n : S_i \cap T_i \neq \emptyset$; *otherwise it is undefined.*

Algorithm 1 merge($\mathcal{S}$)

Require: $\mathcal{S}$: set of substitutions (each represented as a multisubstitution)
  while $\exists u, v \in \mathcal{S}$ such that $u \neq v$ and $u \sqcup v = t$ **do**
    $\mathcal{S} := (\mathcal{S} \setminus \{u, v\}) \cup \{t\}$
  return $\mathcal{S}$

Example 6. The intersection of $\Sigma = \{X \to \{1,3,4\}, Z \to \{2,8,9\}\}$ and $\Theta = \{Y \to \{7\}, Z \to \{1,2,9\}\}$ is: $\Sigma \sqcap \Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$. The intersection of $\Sigma = \{X \to \{1,3,4\}, Z \to \{8,9\}\}$ and $\Theta = \{Y \to \{7\}, Z \to \{1,2\}\}$ is undefined.

The above $\sqcap$ operator is able to check if two multisubstitutions are compatible (i.e., if they share at least one of the substitutions they represent). Indeed, given two multisubstitutions Σ and Θ, if $\Sigma \sqcap \Theta$ is undefined, then there must be at least one variable X, common to Σ and Θ, to which the corresponding multibinds associate disjoint sets of constants, which means that it does not exist a constant to be associated to X by both Σ and Θ, and hence a common substitution cannot exist as well. The $\sqcap$ operator can be extended to the case of sets of multisubstitutions. Specifically, given two sets of multisubstitutions $\mathcal{S}$ and $\mathcal{T}$, their intersection is defined as the set of multisubstitutions obtained as follows:

$$\mathcal{S} \sqcap \mathcal{T} = \{\Sigma \sqcap \Theta \mid \Sigma \in \mathcal{S}, \Theta \in \mathcal{T}\}$$

Note that, whereas a multisubstitution (and hence an intersection of multisubstitutions) is or is not defined, but cannot be empty, a set of multisubstitutions can be empty. Hence, an intersection of sets of multisubstitutions can be empty (which happens when all of its composing intersections are undefined).

Proposition 1. *Let* $C = \{l_1, \ldots, l_n\}$ *and* $\forall i = 1 \ldots n : \mathcal{T}_i = merge(uni(C, l_i, D))$*; let* $\mathcal{S}_1 = \mathcal{T}_1$ *and* $\forall i = 2 \ldots n : \mathcal{S}_i = \mathcal{S}_{i-1} \sqcap \mathcal{T}_i$*.* C *θ-subsumes* D *iff* $\mathcal{S}_n \neq \emptyset$*.*

This result, whose proof we omit, leads to the θ-subsumption procedure reported in Algorithm 2. Note that the set of multisubstitutions resulting from the merging phase could be not unique. In fact, it may depend on the order in which the two multisubstitutions to be merged are chosen at each step. The presented algorithm does not specify any particular principle according to which performing such a choice, but this issue is undoubtedly a very interesting one, and deserves a specific study (that is outside the scope of this paper) in order to understand if the quality of the result is affected by the ordering and, in such a case, if there are heuristics that can suggest in what order the multisubstitutions to be merged have to be taken in order to get an optimal result.

Example 7. Given the following substitutions: $\theta = \{X \leftarrow 1, Y \leftarrow 2, Z \leftarrow 3\}$, $\delta = \{X \leftarrow 1, Y \leftarrow 2, Z \leftarrow 4\}$, $\sigma = \{X \leftarrow 1, Y \leftarrow 2, Z \leftarrow 5\}$, $\tau = \{X \leftarrow 1, Y \leftarrow$

Algorithm 2 matching(C, D)

Require: $C : c_0 \leftarrow c_1, c_2, \ldots, c_n$, $D : d_0 \leftarrow d_1, d_2, \ldots, d_m$: clauses
  if $\exists \theta_0$ substitution such that $c_0\theta_0 = d_0$ **then**
    $S_0 := \{\theta_0\}$;
    for $i := 1$ to n **do**
      $S_i := S_{i-1} \sqcap merge(uni(C, c_i, D))$
  return $(S_n \neq \emptyset)$

$5, Z \leftarrow 3\}$ one possible merging sequence is $(\theta \sqcup \delta) \sqcup \sigma$, that prevents further merging τ and yields the following set of multisubstitutions:
$\{\{X \leftarrow \{1\}, Y \leftarrow \{2\}, Z \leftarrow \{3,4,5\}\}, \{X \leftarrow \{1\}, Y \leftarrow \{5\}, Z \leftarrow \{3\}\}\}$
Another possibility is first merging $\theta \sqcup \tau$ and then $\delta \sqcup \sigma$, that cannot be further merged and hence yield:
$\{\{X \leftarrow \{1\}, Y \leftarrow \{2,5\}, Z \leftarrow \{3\}\}, \{X \leftarrow \{1\}, Y \leftarrow \{2\}, Z \leftarrow \{4,5\}\}\}$

Considering that the proposed algorithm yields all the possible substitutions by which θ-subsumption holds, its time performance, even on hard problems, turns out to be in most cases comparable, and in any case at least acceptable, with respect to Django. The experimental results confirming such a claim are not reported in this paper due to lack of space.

4 Resolution

The effectiveness of the new algorithm can be appreciated by considering that it allows to perform resolution between Horn Clauses, avoiding backtracking. This is possible thanks to its feature of returning all the substitutions for a matching problem. Algorithm 3 shows how the matching procedure has to be modified in order to perform resolution. First, the head of the example must be separated from its body (the "observation"): the former will represent the top level goal of the resolution process and will be used only once (at the beginning); the atoms in the latter must be available at each resolution step. Note that the goal of resolution must not be necessarily ground, which allows to use the given procedure also to get computed answer substitutions [6] for unbound variables.

The algorithm has two basic behaviours. The former regards the proof of *basic literals*, i.e. literals that don't have a definition in the theory, but that can be proved using only literals in the example. In this case, all the possible substitutions between the literal to be proven and those in the observation are collected and represented by a single multisubstitution. The latter behaviour concerns the literals built on predicates that have a definiton in the theory. Whenever a literal of this kind is encountered, we are interested in all the possible substitutions coming from any rule that defines the corresponding predicate. Hence, for each such rule, a new child matching process is started on each of its literals and the corresponding results are intersected. The outer loop collects all possible substitutions that make true g with respect to O. Recursive clauses in the theory would lead to non-termination. This can be avoided by handling recursive definitions in a slightly different way, i.e. considering only one clause at a time and performing backtracking just like Prolog does.

Example 8. Given the following theory T:

```
h(X) :- p(X,Y), q(Y,Z), t(X,Z). % 1
p(X,Y) :- g(X,Y), s(Y).         % 2
t(X,Y) :- f(X,Y).               % 3
t(X,Y) :- d(X,Z), t(Y,Z).       % 4 (recursive clause)
```

and the following example E:

```
h(1) :- g(1,2), g(1,3), g(1,4), s(2), s(3), f(4,5),
        q(2,3), q(3,4), q(3,5), d(1,2), d(1,5), d(1,4).
```

The algorithm chooses from T a clause C whose head is unifiable with the head of E (if any, otherwise it fails). Hence, it chooses clause (1) (with the multisubstitution {{X→{1}}}) and begins to prove it by selecting $p(X, Y)$. To solve $p(X, Y)$ clause (2) is choosen, that is verified by the multisubstitution {{X→{1},Y→{2,3}}}. Indeed, literal $g(X, Y)$ yields the multisubstitution {{X→{1}, Y→{2,3,4}}}, that intersected with the multisubstitution associated to literal $s(Y)$, {{Y→{2,3}}}, gives the final multisubstitution for clause (2) (note that the multisubstitutions for the literals $g(X, Y)$ and $s(Y)$ are obtained using literals in E because there are no clauses for these predicates in T). Now, the algorithm must prove literal $q(Y, Z)$, that is true for {{Y→{2}, Z→{3}}, {Y→{3}, Z→{4,5}}} by exploiting literals in E. The partial multisubstitution {{X→{1}, Y→{2}, Z→{3}}, {X→{1}, Y→{3}, Z→{4,5}}} for clause (1) is obtained. Finally, to prove $h(1)$ it is necessary to prove literal $t(X, Z)$ in clause (1). Predicate t is recursively defined in the theory (clauses (3) and (4)), thus the algorithm selects first clause (3), that is true for {{X→{4}, Y→{5}}} but incompatible with the partial multisubstitution obtained so far. Then, it selects clause (4), it solves literal $d(X, Z)$ with {{X→{1}, Z→{2,4,5}}} and then calls recursively $t(Y, Z)$, that returns (using this time clause (3)) the multisubstitution {{Y→{4}, Z→{5}}}. Finally, the algorithm returns for literal $t(X, Z)$ in clause (1) the multisubstitution {{X→{1}, Z→{4}}} that is compatible with the partial one yielding the final multisubstitution {{X→{1}, Y→{3}, Z→{4}}}. This process proves that $h(1)$ is true in T via resolution.

5 Conclusions and Future Work

This paper proposed a new algorithm for computing θ-subsumption, that is able to carry out its task with high efficiency and can be extended in order to

Algorithm 3 Resolution

Resolution(g,T,O):S
Given a goal g, a theory T and an observation O
if g is unifiable with some literal in O **then**
  $S \leftarrow merge(uni(C, g, O))$
else
  $S \leftarrow \emptyset$
  for all clauses $C \in T$ such that head(C) and g are unifiable **do**
    $C \leftarrow C\theta$, s.t. $head(C)\theta = g$
    $S' \leftarrow merge(\theta)$ {applies only to constants in θ}
    for each literal $l \in body(C)$ **do**
      $S' \leftarrow S' \sqcap$ Resolution(l,T,O)
  $S \leftarrow S \sqcup S'$
return(S)

perform resolution. A Prolog version of the extended algorithm is currently used in the ILP system *INTHELEX* [3]. Future work will concern an analysis of the complexity of the presented algorithm, and the definition of heuristics that can further improve its efficiency.

Acknowledgements. This work was partially funded by the EU project IST-1999-20882 COLLATE. The authors are grateful to M. Sebag and J. Maloberti for making available their system Django, and for the kind suggestions on its use and features.

References

[1] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[2] N. Eisinger. Subsumption and connection graphs. In J. H. Siekmann, editor, *GWAI-81, German Workshop on Artificial Intelligence, Bad Honnef, January 1981*, pages 188–198. Springer, Berlin, Heidelberg, 1981.

[3] F. Esposito, G. Semeraro, N. Fanizzi, and S. Ferilli. Multistrategy Theory Revision: Induction and abduction in INTHELEX. *Machine Learning Journal*, 38(1/2):133–156, 2000.

[4] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295, 1985.

[5] J.-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In W. Cohen and H. Hirsh, editors, *Proc. Eleventh International Conference on Machine Learning (ML-94)*, pages 130–138, 1994.

[6] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.

[7] J. Maloberti and M. Sebag. θ-subsumption in a constraint satisfaction perspective. In Céline Rouveirol and Michèle Sebag, editors, *Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France*, volume 2157, pages 164–178. Springer, September 2001.

[8] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, January 1965.

[9] T. Scheffer, R. Herbrich, and F. Wysotzki. Efficient θ-subsumption based on graph algorithms. In Stephen Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming (ILP-96)*, volume 1314 of *LNAI*, pages 212–228, Berlin, August 26–28 1997. Springer.

[10] M. Schmidt-Schauss. Implication of clauses is undecidable. *Theoretical Computer Science*, 59:287–296, 1988.

[11] R.B. Stillman. The concept of weak substitution in theorem-proving. *Journal of ACM*, 20(4):648–667, October 1973.