A simple framework for theta-subsumption testing in Prolog

Hendrik Blockeel[1], Svetlana Valevich[2]

[1] Department of Computer Science, KU Leuven
[2] KU Leuven

Abstract. We present a simple framework for theta-subsumption testing in Prolog. In its simplest instantiation, it yields an algorithm that takes only a few dozen lines of code. Despite its simplicity, the framework has turned out to work very well on data where a state-of-the-art subsumption engine suffered from excessive run times. The framework can easily be instantiated in different ways, precisely because of its simplicity, and can offer an interesting view on how existing methods compare to each other.

1 Introduction

Theta-subsumption testing is a crucial and ubiquitous operation in ILP. It is used in the context of computing the coverage of clauses, but also when reducing a clause (finding the shortest subclause equivalent to it), which is an important step in computing the least general generalization of two clauses.

Theta-subsumption testing is NP-hard. Several advanced methods have been proposed that try to test it efficiently under a wide range of conditions. For most of these, an implementation is available. These implementations are often complex, which makes it difficult to understand them fully even if source code is available. They are also written in a variety of programming languages, making integration into a Prolog program nontrivial.

In this paper, we propose a simple framework for theta-subsumption testing in Prolog. The framework has the following advantages: (1) it is easy to understand; (2) the simplest algorithm within this framework takes only a few dozen lines of Prolog code and is already quite efficient for a wide range of clauses; (3) the framework offers a useful view on how multiple existing methods relate to each other; (4) the framework makes a link with query execution in databases.

2 Theta-subsumption

2.1 Definition

We assume familiarity with standard terminology from logic programming and relational databases, and with the Prolog programming language. Theta-subsumption, or briefly subsumption, is defined as follows: 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 $c\theta \subseteq d$.

2.2 Testing theta-subsumption in Prolog

We assume clauses are represented as lists of literals. The following code can then be used to test whether a clause C subsumes a clause D:

```
subsumes(C,D) :-
    \+ \+ (copy_term(D, D2), numbervars(D2, 0, _), subset(C, D2)).

subset([], D).
subset([A|B], D) :- member(A, D), subset(B,D).
```

The first clause makes a copy D_2 of D to rename its variables apart from those in C, then skolemizes D_2 (that is, instantiates each variable in it to a different constant), and finally tries to unify C with a subset of D_2. The skolemization of D_2 is necessary to avoid that the unification procedure applies variable substitutions to both C and D, instead of only to C. The double negation (`\+ \+`) ensures that the unifications do not survive the call, so that `subsumes(C,D)` does not have any side effects.

The above code exploits Prolog's unification and backtracking mechanisms. A call to `member(A,D)` may have multiple solutions, requiring different substitutions for the variables in A. Some of these substitutions may make other literals (which share variables with A) un-unifiable with any member of D. This may be noticed only later on; Prolog then has to backtrack and try a different substitution for A.

When the first clause contains variables, the subset predicate defines a search through the space of all possible variable substitutions. The size of this space is the product of the number of possible instantiations for each variable, and hence exponential in the number of variables occurring in C.

2.3 Partitioning into independent components

A simple way to make the search more efficient is to first partition the clause C into minimal subsets such that different subsets share no variables. The choice of a substitution for one subset then cannot affect the existence of a substitution for another subset. Each subset can then be tested separately using the code listed above. The complexity of this method is exponential in the largest number of variables in any one subset (as opposed to the total number of variables).

Example 1. Let $C = \{p(X), q(X,Y), r(Z,2))\}$ and $D = \{p(1), p(2), p(3), p(4), q(2,a), q(4,b), r(b,1)\}$. After finding a substitution for `p(X)` and `q(X,Y)`, Prolog tries to find a substitution for `r(Z,2)`. There is none. Prolog will then backtrack, trying to find a different substitution for X and Y. But it is clear that none of these alternatives will change the fact that no substitution exists for Z that makes `r(Z,2)` an element of D. Alternative solutions for X and Y do not affect the existence of solutions for Z. Therefore, it is more efficient to split C into $C_1 = \{p(X), q(X,Y)\}$ and $C_2 = \{r(Z,2)\}$ and perform the search for each of these separately. The complexity of this is $O(|S_{XY}|+|S_Z|)$ instead of $O(S_{XYZ}) = O(|S_{XY}| * |S_Z|)$.

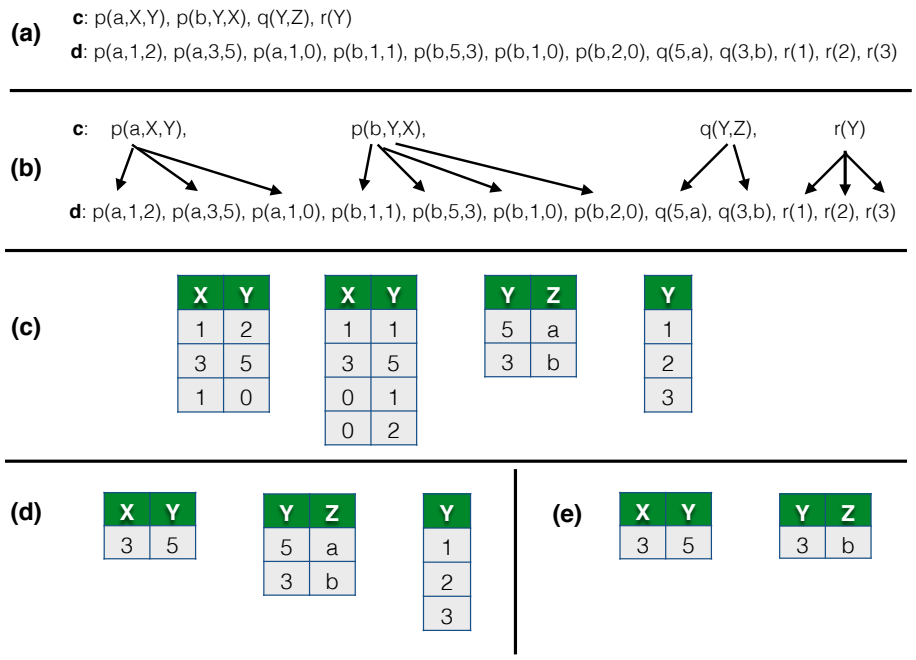

Fig. 1. Constructing instantiation tables. (a) Clauses c and d for which $c \leq_\theta d$ is to be tested. (b) For each literal in c, the matching literals in d are shown. (c) For each literal l in c, an instantiation table shows the possible instantiations of the variables notation in l. (d) Two tables with the same attributes (variables) can be replaced by their intersection. (e) When the attributes of table T are a subset of the attributes of T', T' can be dropped after replacing T by $T \ltimes T'$.

The splitting of clauses into independent parts, each solved separately, is a crucial element in all efficient theta-subsumption testers.

2.4 Pruning the search space based on semi-joins

In the context of testing $C \preceq_\theta D$, let d be the skolemized version of D and c one of the independent subsets of C. A *solution* is a variable substitution θ such that $c\theta \subseteq d$. A literal l_1 *matches* a literal l_2 if and only if a variable substitution θ exists such that $l_1\theta = l_2$. The *instantiation list* of a literal $l \in c$ is the set of all literals $l' \in d$ that match l. Figure 1 shows an example pair of clauses c and d (a), and for each literal in c the matching literals in d (b).

Using terminology from relational databases, we define the *instantiation table* of a literal l as a table with as attributes the variables in l, and as tuples the value combinations for these variables that occur in the instantiation list. An instantiation table is simply a different representation for an instantiation list. Figure 1(c) shows the instantiation tables of the four literals in c.

Note that the instantiation tables serve as constraints on the possible instantiation of tuples of variables. Let T be an instantiation table and $\mathbf{X}$ the set of variables corresponding to its attributes. Any solution θ must instantiate $\mathbf{X}$ such that $\mathbf{X}\theta$ occurs in T. Therefore:

- when tables T and T' have the same attribute set, they can be replaced by a single table that contains their intersection.
- more generally, when the attributes of T are a subset of those of T', $attr(T) \subseteq attr(T')$, all tuples t' of T' for which $\pi_{attr(T)}(t') \notin T$ can be removed from T', and after doing this, T can be dropped (it no longer imposes a constraint that is not already imposed by T')
- even more generally, let $A = attr(T) \cap attr(T')$; when $A \neq \emptyset$, all tuples t' of T' for which $\pi_A(t') \notin \pi_A(T)$ can be removed from T', and vice vera, all $t \in T$ for which $\pi_A(t) \notin \pi_A(T')$ can be removed from T.

Figure 1 illustrates the effect of replacing the first two tables by their intersection (d), and that of dropping the Y table after filtering the YZ table based on it (e).

In relational algebra terms, the above three rules can be summarized as follows. (1) A table T can always be replaced by $T \ltimes T'$, its semi-join with T'. The (left) semi-join operator is defined as $T \ltimes T' = \pi_A(T * T')$, with π the projection operator, $*$ the natural join, and A the set of all attributes of T. (2) After this semi-join, T' can be dropped if $attr(T') \subseteq attr(T)$. These operations can reduce the number of tables and the size of each table.

2.5 The search itself

When the search space can no longer be reduced in the way described, an exhaustive search is needed. The following procedure is then recursively applied: choose a table T; for each tuple $t \in T$: filter the other tables by leaving out all tuples incompatible with t (that is, remove each $t' \in T'$ such that $\pi_A(t') \notin \pi_A(T)$ with $A = attr(T) \cap attr(T')$); call the search procedure recursively on the resulting set of tables. When there are no tables left to choose from, a solution has been found. If at any point, a table becomes empty, the search must backtrack and choose the next t; if no alternatives for t are left, this means no solutions exist.

The selection of T can be done according to a heuristic. Ideally, T is chosen such that it reduces the search space as much as possible. For a table with n attributes, each with domain size m, the table's domain size m^n. Among all values in this domain, only the tuples in the table are valid. Thus, if the table contains l tuples, the search space is reduced by a factor m^n/l (compared to exhaustively trying all value combinations). This factor can be used as a heuristic. Another factor to take into account is: in how many other tables lists do the attributes occur, and to what extent will those tables be reduced? Finally, as observed by Santos and Muggleton [5], the instantiation may cause further decomposition of the clause. A heuristic that tries to maximize such decomposition is likely to be advantageous.

2.6 Algorithm

The above points give rise to a simple algorithm for efficient subsumption testing. A number of auxiliary functions and procedures of the algorithm can be instantiated in different manners. The algorithm thus gives rise to a framework that we an easily experiment with.

Our current implementation uses a very simple and rough heuristic: it simply uses $5^n/l$ (that is, it assumes an average domain size of 5). It performs semi-joins until a fixpoint is reached. The filtering during the search uses a single step of semi-joining the chosen tuple with the other tables. Decomposition into independent components is done only at the beginning, not after each instantiation.

2.7 Situating existing methods in this framework

Existing subsumption algorithms can be situated within this framework, making it easier to explain and compare them. We here focus on Subsumer [5], Resumer [3], Django [4], and the algorithm proposed by Scheffer, Herbrich and Wysotzky [6], which we call SHW here. For lack of space, we focus on two main differences here.

All methods mentioned above phrase the problem as a constraint programming problem and use advanced constraint solving methods. An important difference among methods is in how they define the variables and domains for the solver. Subsumer and Resumer use as variables the **logical variables** that occur in the clauses. Django and SHW use as variables the **literals** in c, and as possible values the (matching) literals in d. From the constraint solving point of view, our method uses **tuples of logical variables** as the variables to solve for, and tuples of values as their values. The use of single variables is inherently less efficient, as information about which combinations of values occur in clause d is ignored. Using variable tuples as opposed to literals has the advantage that when the same tuple of variables occurs in multiple literals, only one variable is introduced for them in the constraint solver.

Instantiating a variable tuple may cause a connected component to decompose into independent components. Subsumer is the only method to exploit this. Santos and Muggleton showed that this can yield important efficiency gains [5]. The idea can easily be incorporated into our framework. The selection heuristic can be adapted to take the ensuing search space reduction into account.

3 Experimental Evaluation

This work was motivated by work on learning language from sentences in a context, where the context is described as a logical interpretation [1, 2]. The incremental learning algorithm proposed there repeatedly computes lggs, which themselves involve multiple subsumption tests, and the straightforward implementation of subsumption testing quickly turned out too slow. Somewhat to our surprise, also Subsumer, the most recently proposed subsumption engine, turned

out to be problematically slow on some cases. Repeated attempts to improve our own straightforward subsumption algorithm yielded a simple instantiation of the framework described above.

We compared the runtimes of Subsumer and our algorithm, dubbed Subtle,[3] on a dataset of 10000 sentence/context examples. Due to the NP-hardness of the problem, average timings take very long to obtain and are not very informative (they are strongly influenced by the heavy tails of the runtime distribution). We have therefore followed the following methodology. The incremental learner processes examples one by one. When the processing of a single example takes over a minute (indicating that it gave rise to a "difficult" pair of clauses), the example is commented out and the learner restarted. As more and more such "problematic" examples are removed, the learner gets further into the dataset before getting stuck. After removing 10 problematic examples, we stopped.

Subsumer had its 10th problematic example at index 73. Subtle had only 7 problematic examples; the remaining 9993 were processed in 27 minutes.

On one benchmark included in the Subsumer distribution (specifically, testing 40 hypotheses from the file `hyp1_00.pl` on the 400 examples in `exs_00.pl`), we found CPU time ratios for Django, Subsumer and Subtle of roughly 1:5:30. On separate sets of subsumption problems generated from the language learning problem, we found the ratios of roughly 1: – : 5 on one dataset, and – : 7 : 12 on another dataset, where – indicates that the system did not run to completion (due to memory problems or excessive time). While Subtle is not the fastest method on either dataset, it is the only one that could handle both.

All this indicates that different methods are best for different datasets, and a versatile framework is therefore useful. We believe that our simple framework can offer that versatility, but more experiments are needed to confirm this.

4 Conclusions

We have proposed an framework for testing theta-subsumption that is easy to understand, easy to implement in Prolog, and significantly outperforms a state-of-the-art system on a practically motivated problem. Given the NP-hardness of theta subsumption, it is to be expected that different algorithms will be optimal under different circumstances. An important advantage of our framework is that it can easily be adapted to different types of datasets. The simplicity of the framework also makes it possible to incorporate ideas from different existing systems into it.

A more extensive experimental evaluation is needed to evaluate the true potential of the method. This will include experiments on a variety of datasets, and with different instantiations of the method.

[3] "SUBsumption Testing with Little Effort"

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. Kuzelka, O., Zelezný, F.: A restarted strategy for efficient subsumption testing. Fundam. Inform. 89(1), 95–109 (2008)
4. Maloberti, J., Sebag, M.: Fast theta-subsumption with constraint satisfaction algorithms. Machine Learning 55(2), 137–174 (2004)
5. Santos, J., Muggleton, S.: Subsumer: A prolog theta-subsumption engine. In: Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK. pp. 172–181 (2010)
6. Scheffer, T., Herbrich, R., Wysotzki, F.: Efficient theta-subsumption based on graph algorithms. In: Inductive Logic Programming, 6th International Workshop, ILP-96, Stockholm, Sweden, August 26-28, 1996, Selected Papers. pp. 212–228 (1996)