Tabling with Answer Subsumption: Implementation, Applications and Performance

Terrance Swift[1] and David S. Warren[2]

[1] CENTRIA — Universidade Nova de Lisboa
[2] Stony Brook University, Stony Brook, NY

Abstract. Tabled Logic Programming (TLP) is becoming widely available in Prolog systems, but most implementations of TLP implement only *answer variance* in which an answer A is added to the table for a subgoal S only if A is not a variant of any other answer already in the table for S. While TLP with answer variance is powerful enough to implement the well-founded semantics with good termination and complexity properties, TLP becomes much more powerful if a mechanism called *answer subsumption* is used. XSB implements two forms of answer subsumption. The first, partial order answer subsumption, adds A to a table only if A is greater than all other answers already in the table according to a user-defined partial order. The second, lattice answer subsumption, may join A to some other answer in the table according to a user-defined upper semi-lattice. Answer subsumption can be used to implement paraconsistent and quantitative logics, abstract analysis domains, and preference logics. This paper discusses the semantics and implementation of answer subsumption in XSB, and discusses performance and scalability of answer subsumption on a variety of problems.

1 Introduction

Tabled Logic Programming (TLP) currently supports a number of applications in agent frameworks, reasoning over the semantic web, machine learning, and probabilistic logic programming; and TLP is supported by several Prolog systems, including XSB, YAP, B Prolog, Ciao, and ALS. However, an important feature called *answer subsumption* has been little studied in the literature, and is missing from most TLP systems. Most TLP systems add an answer A to a table T only if A is not a variant of some other answer already in T, a technique termed *answer variance*. While answer variance is sufficient to allow tabling to compute the well-founded semantics and to terminate for programs with bounded term-depth, other choices of when and how to add an answer can be made. Using *partial order answer subsumption*, A would be added to T only if A is maximal with respect to other answers in T according to a given partial order $>_O$. Furthermore if A is added, any answers in T that A subsumes (i.e., is greater than in $>_O$) are deleted. When using *lattice answer subsumption*, A itself may not be added to T, rather the join is taken of A and another answer A' in T, with A' being deleted. Despite its conceptual simplicity, answer subsumption can be a powerful tool. Partial order answer subsumption allows a table to retain only answers that are maximal according to a metric or to a preference relation; lattice answer subsumption can form

the basis of multi-valued logics, quantitative logics, and of abstract interpretations for programs and process logics.

A version of answer subsumption has been available in XSB for over a decade, but its implementation was never described, and only recently was its implementation optimized and declarations provided to make it easy for programmers to use. [10] described how lattice answer subsumption can implement Generalized Annotated Programs [6], but did not provide any details of implementation or benchmarks. Recently, [8] used answer subsumption to implement probabilistic inference, but the benchmark times in that paper were dominated by the cost of maintaining BDDs that represented probabilistic explanations. Beyond related work for XSB, the mode-specific tabling of B Prolog can be seen a restricted form of answer subsumption that allows only min and max over the Prolog term order, and constrains the modes of aggregated tabled subgoals.

This paper makes two main contributions: first, it describes the implementation of partial order and lattice answer subsumption in XSB; and second, it analyzes performance and demonstrates scalability of answer subsumption for applications in social network analysis, abstract interpretation, and query justification through multi-valued logics. The structure of the paper is as follows. Section 2 informally presents the semantics of partial order and lattice answer subsumption. Section 3 then describes the underlying implementation of answer subsumption using the trie-based tabling data structures of XSB. Finally, Section 4 analyzes the performance and scalability of answer subsumption in various applications.

2 An Informal Semantics for Answer Subsumption

Terminology and Conventions. Informally, an answer is simply an atom derived via some fixed-point evaluation of a program P – using tabling or bottom-up evaluation. For simplicity of presentation, we assume that all queries are safe – i.e. that any answer to a query will be ground (the implementation in XSB allows non-ground answers in certain cases). Within this paper, answer subsumption is restricted to occur on a single argument of a predicate; however since answer subsumption is defined for arbitrary terms this restriction does not affect expressibility. For purposes of space, we restrict our description to definite programs. However, the implementation described in Section 3 supports stratified programs, and so can form the basis of formalisms that use negation such as annotated or residuated programs [6, 1]. Finally, all examples use standard Prolog syntax.

Partial Order Answer Subsumption. For simplicity, our first examples make use of a shortest-path predicate (Figure 1) that counts the number of edges between two vertices; more sophisticated uses of answer subsumption are presented in Section 4.

```
sp(X,Y,1):- edge(X,Y).
sp(X,Z,N):- sp(X,Y,N1),edge(Y,Z),N is N1 + 1.
```

Fig. 1. A Shortest Path Predicate

As mentioned above, partial-order answer subsumption retains in a table T only those answers that are maximal according to a given partial order $>_O$. In the case of the shortest-path predicate of Figure 1, $sp(A_1, A_2, A_3) >_O sp(B_1, B_2, B_3)$ if, $A_1 = B_1$, $A_2 = B_2$, and $A_3 < B_3$. Note that that minimal distances are maximal in $<_O$, and that $<_O$ is undefined if A_3 or B_3 is non-numeric. In XSB, partial order answer subsumption is specified for `sp/3` using the declaration

```
:- table sp(_,_,po(</2)).
```

In a given state of computation, only those answers that are maximal according to $>_O$ are available for resolution. Thus, for a finite graph with cycles, `sp/3` will terminate using answer subsumption, but not with answer variance. Other partial orders beyond distance metrics may be useful. For instance, $>_O$ may specify a preference ordering between derived atoms so that answer subsumption provides an alternative to default-based methods for computing preferences (cf. Section 4.1 for a discussion).

Lattice Answer Subsumption. An upper semi-lattice is a partial order for which any two elements have a unique least upper bound. Because the ordering for the third argument of sp/3 is total, it also forms an upper semi-lattice, and so can be computed using lattice answer subsumption. In XSB lattice answer subsumption for `sp/3` is declared as

```
:- table sp(_,_,lattice(min/3)).
```

with `min/3` defined as `min(X,Y,Z):- Z is min(X,Y).` Operationally, this means that whenever an answer $sp(A_1, A_2, A_3)$ is derived, if there is another answer $sp(B_1, B_2, B_3)$ where $A_1 = B_1$ and $A_2 = B_2$ the join J_3 of A_3 and B_3 is taken, and only $sp(A_1, A_2, J_3)$ is available for resolution. As with a partial order, the join operation ensures termination for shortest path over a finite graph with cycles.

As the following proposition shows, lattice answer subsumption can be modeled either starting with a lattice, or starting with a function with appropriate properties.

Proposition 1. *Let* op *be an associative, commutative, and idempotent binary function. Then there is a partial order P, such that P is an upper semi-lattice with join* op.

Conversely, if a function does not have the above properties, it is not suitable for lattice answer subsumption. Accordingly the aggregate functions count and sum cannot be computed using lattice answer subsumption [3]. Lattice answer subsumption has a variety of applications: Section 4.3 shows how it is used for social-network analysis and for an application of multi-valued logics, [10] describes how a similar formalism can implement a quantitative logic, and [8] describes an implementation of probabilistic logic based on answer subsumption.

Partial Order Answer Subsumption with Abstraction. Computation over an abstract domain may require certain maximal answers to be abstracted. In many cases, abstraction can be modeled by a join operation, but in others the abstraction represents an implicit induction step in the following sense. Given a set $\mathcal{A}$ of answers, it may be detected that

[3] Since count and sum are not idempotent their semantics is based on multi-sets, rather than sets. Incorporating these as tabling features requires modifying their semantics to be set-based, in a manner similar to aggregation ASP systems (cf. e.g. [2]) .

the program computed does not have a finite model. An abstraction operation then is applied so that $\mathcal{A}$ and its extensions can be symbolically represented by a single answer A. Using answer subsumption, this abstraction can be taken only if needed during program execution. Abstractly, partial order answer subsumption with abstraction uses the declaration

```
:- table p(_,_,po(rel/2,abs/3)).
```

where `rel/2` is a partial order, and `abs/3` is the abstraction operation. Section 4.2 provides a detailed example of how such an approach is used to analyze a process logic.

Complexity. Consider a ground program P where some predicate p/n is declared to use lattice answer subsumption with join predicate $op/3$. Note that any answer to a subgoal of p/n need be compared to at most one other answer to compute a join. Thus if $op/3$ has constant cost, lattice answer subsumption adds no overhead in terms of complexity to evaluating P. However, for partial order answer subsumption, an answer to a subgoal of p/n might in principle be compared to all other answers for p/n, which in the worst case is $atoms(P)$, the number of atoms in P. Accordingly, if $rel/2$ has constant cost, the complexity of evaluating P will be $size(P) \times atoms(P)$, regardless of whether P is definite, or is being evaluated using negation over the well-founded semantics.

3 Implementation

Both lattice and partial-order subsumption are implemented through a compiler translation that introduces specialized code to manipulate answers in the table.

We first describe the implementation of lattice answer subsumption. As discussed, for simplicity of presentation, we assume that the predicate tabled using answer subsumption returns only ground answers. Consider again the example of shortest path using `min/3` as a join operator (Figure 1), in which the query finds all distances from a single source – e.g. a query such as `sp(a,Y,M)`. The XSB compiler transforms `sp/3` to the code in Figure 2. The first two subgoals in the body of the transformed version of `sp/3` (line 3) gain access to the table created on the call to `sp(a,Y,M)`; access in XSB is through the generator choice point for the table, obtained through the choice point register, `Breg` (see [9] for details). `Cs` is a pointer to the table entry for the current call, and `Skel` is a term containing the free variables of the query, which for `sp(a,Y,M)` is the term `ret(Y,M)`. Since tabled answers in XSB contain only bindings to variables in the call, the free variables are necessary to retrieve answers from the table. Line 4 throws an error if the argument using answer subsumption is not a variable, as the code of Figure 2 is not correct in that case. Lines 5 and 6 generate variants of other terms that will be needed to retrieve answers from the table. In our example, `OSkel` is `ret(Y,OM)` – note that `Y` is in the call, but `OM` is free. After this setup, line 7 calls the original code (transformed to `'sp$$'/3`) to derive answers. On success of `'sp$$'/3` (line 8), a previous answer whose bindings unify with `OSkel` is obtained from the table, if it exists. For instance, if the success of `sp$$/3` in line 7 bound `Y` to `b`, the answer in the table for `sp(a,Y,M)` that has `Y` bound to `b` is obtained if it exists, binding `OM` to the third argument of that answer. Note that the use of lattice

```
   :- table sp/3.
   sp(X,Y,M) :-
       '_$savecp'(Breg), breg_retskel(Breg,3,Skel,Cs),
       (nonvar(M) -> instantiation_error ; true),
5      excess_vars(Skel,[M],[],Vars),
       copy_term(t(Vars,Skel,M),t(Vars,OSkel,OM)),
       'sp$$'(X,Y,NM),
       ('_$$get_answers'(Cs,OSkel,AnsPtr)
        -> min(OM,NM,M),
10          M \== OM,
            delete_answer(Cs,AnsPtr)
        ;   M = NM ).

   'sp$$'(X,Y,1) :- edge(X,Y).
15 'sp$$'(X,Y,N) :- sp(X,Z,N1),e(Z,Y),N is N1+1.
```

Fig. 2. Example Code for Lattice Answer Subsumption

answer subsumption, together with the safety assumption ensure that there is at most one such answer. If the answer does exist, the old value `OM` is joined with the new `NM` from the answer just returned (line 9). If the join differs from the old answer (line 10), the old answer is deleted (line 12) and the clause succeeds. Further compilation into byte code ensures that an answer is added to the table whenever a clause of a tabled predicate succeeds (here, in line 10 or 12). If the joined value `M` is the same as the value `OM` in the old answer, the computation fails in order to search further. If there is no previous answer in the table (line 12), then the clause succeeds. Note that the setup portion, (lines 1-6) are executed once per call; lines 8-12 are executed for each answer.

```
  :- table sp/3.
  sp(X,Y,M) :-
      '_$savecp'(Breg), breg_retskel(Breg,3,Skel,Cs),
      (nonvar(M) -> instantiation_error ; true),
5     excess_vars(Skel,[M],[],Vars),
      copy_term(t(Vars,Skel,M),t(Vars,OSkel,OM)),
      'sp$$'(X,Y,NM),
      \+ ('_$$get_answers'(Cs,OSkel,_),
          (OM == NM ; '<'(OM,NM)) ),
10    findall(AnsPtr,
              ('_$$get_answers'(Cs,OSkel,AnsPtr), '<'(NM,OM)),
              AnsPtrs),
      (member(AnsPtr,AnsPtrs), delete_answer(Cs,AnsPtr), fail
       ;
15     M=NM ).
```

Fig. 3. Example Code for Partial Order Answer Subsumption

Next we describe the implementation of the same program and query using partial order answer subsumption. Again the compiler transforms the program to perform the table manipulations (Figure 3). The first 6 lines of setup are identical to the lattice case; partial order subsumption differs only in how it treats answers. In lines 8-9 the table is checked to see if any previous answer is the same as or subsumes the new answer. If so, then the computation fails. (Note that if the new answer is subsumed by an answer already in the table, then the table will not contain any answer subsumed by the new one.) Assuming the new answer is not subsumed by any old answer, lines 10-12 use `findall/3` to collect pointers to all answers subsumed by the new one, and in line 13, they are deleted from the table. The new answer is added to the table upon the clause's success in line 15.

Finally we describe the transformation for Partial Order Subsumption with Abstraction. The example transformation for PT Net Reachability (Section 4.2) is shown in Figure 4. The declaration for this example is assumed to be `:- table reachable(_,po(omega_gte/2,omega_abs/3)).` Again the

```
  :- table reachable/2.
  reachable(S,M) :-
      '_$savecp'(Breg), breg_retskel(Breg,2,Skel,Cs),
      (nonvar(M) -> instantiation_error ; true),
5      excess_vars(Skel,[M],[],Vars),
      copy_term(t(Vars,Skel,M),t(Vars,OSkel,OM)),
      'reachable$$'(S,NM),
      findall(OM-AnsPtr,'_$$get_answers'(Cs,OSkel,AnsPtr),OldAnswerPtrs),
      collect_ans(OldAnswerPtrs,OldAnswers),
10     omega_abs(OldAnswers,NM,AbsM),
      \+ (member(OM-_,OldAnswerPtrs),
          (OM == AbsM ; omega_gte(OM,AbsM)) ),
      (member(p(OM,AnsPtr),OldAnswerPtrs),
       omega_gte(AbsM,OM)), delete_answer(Cs,AnsPtr), fail
15     ;
       M=AbsM ).
```

Fig. 4. Example Code for Partial Order Answer Subsumption with Abstraction

setup and call in line 7 are the same as the previous cases. On return of a newly computed answer, line 8 collects all old answers and the pointers to them, and line 9 separates out just the old answers, which are input to the abstraction operator in line 10. Then in the rest of the code, the abstracted answer is used in place of the computed answer, as follows. First, lines 11-12 check whether the new answer is already subsumed by an existing answer, in which case the clause fails. Otherwise, lines 13-14 delete all old answers subsumed by (the possible abstraction of) the new answer. And in line 15, we return the new (possibly) abstracted answer.

4 Performance and Applications

In this section we benchmark and analyze application TLP programs. [4]

4.1 Answer Subsumption in Support of Social Network Analysis

The field of Social Network Analysis (SNA) (cf. [13]) studies the behavior of groups through the relations among their members. In SNA a social network is a graph that is analyzed to determine measures of connectivity or of balance, partitioned into subcomponents according to an optimality criterion, or analyzed in other ways. Logic programming offers promise for SNA: it is easy to specify properties of vertices ("male,"lives-in-city") and of edges ("father-of","exchanges-needles-with"); and SNA properties can be declaratively analyzed by TLP or ASP systems. A factor in many types of SNA (e.g. [11]) is the *coherence* of a (sub-)graph: a numeric measure based on the shortest paths between all vertices in the subgroup (the metric for distance may be defined on different edge types, or their combination).

We begin our benchmarking with the shortest path predicate `sp/3` of Figure 1 which uses lattice answer subsumption. In `sp/3`, distance between two vertices is defined simply as the minimal number of edges between them. While there are several well-known algorithms to determine shortest paths in graphs with non-negative edge weights, the problem offers excellent scope for analyzing various aspects of answer subsumption. Table 1 shows the scalability of the goal `sp(From,To,Dist)` on randomly generated graphs with N vertices and edges. These graphs are sparse in the sense that they are largely unconnected: the number of answers is substantially below the N^2 answers the query would return for a fully connected graph. As Table 1 shows, `sp/3` scales linearly in answers up to the amount of core memory available.

Vertices	Time	Table Space	Answers
25000	1.7	44,146,000	960,588
50000	7.5	198,905,244	4,324,742
75000	12.8	307,611,736	6,683,493
100000	9.8	212,186,848	4,611,563
125000	57.6	1,128,215,852	24,617,754

Table 1. Scalability of lattice `sp/3` on sparse graphs where $|edges| = |vertices|$

The standard algorithm for finding shortest paths to all nodes from a single source node is Dijkstra's algorithm [4]. The difference, between that algorithm and the underlying algorithm for answer subsumption, is in the scheduling. In Dijkstra's algorithm,

[4] All benchmarks were performed on a MacBook pro laptop, with a 2 Ghz Intel Core Duo CPU and 2 GB of RAM. Multi-threading was not used for these benchmarks, so only one core was utilized. All times are in seconds, and all measures of space are in bytes. Table space in XSB includes storage space for subgoals and answers along with space allocated for copying areas, answer hash buckets, etc. All benchmark programs are available by anonymous CVS from xsb.sourceforge.net in the `benches` directory of the module `mttests`.

the next node chosen to expand is the one with shortest distance from the source node. So the "wave front" of the search is expanded by choosing the nearest non-expanded node. This tabling algorithm expands the wave front based on the number of edges from the source, independent of the weights on the edges. For our examples where each edge is assumed of weight 1, the algorithm corresponds to Dijkstra's. But with varying edge weights, answer subsumption (as implemented here) may be suboptimal.

Sparse graphs are unlikely to have many different paths between two vertices: accordingly Table 1 does not check the efficiency of all aspects of lattice answer subsumption such as accessing previously derived answers to compute a join, and possibly deleting them. These factors are measured in Table 2, which benchmarks various predicates on graphs of 1000 vertices and $N = 2 \times 1000, 4 \times 1000 \ldots 512 \times 1000$ edges. In addition to benchmarking `sp/3` with lattice and partial order answer subsumption, Table 2 measures two new predicates shown in in Figure 5. The first, `reach/3` is a simple transitive closure predicate that does the same work as `sp/3` *except* for answer subsumption; the second is a shortest path predicate, `sp_del/3`, for which distance is a function of weights for each edge. As can be seen from Table 2, once the graphs

```
:- table reach/3.
reach(X,Y,1):- edge(X,Y).
reach(X,Z,1):- reach(X,Y,N1),edge(Y,Z), _N is N1 + 1.

:- table sp_del(X,Y,lattice(min/3)).
sp_del(X,Y,D):- edge(X,Y,D).
sp_del(X,Z,D3):- sp_del(X,Y,D1),edge(Y,Z,D2),D3 is D1 + D2
```

Fig. 5. Predicates for shortest path and transitive closure

are fully connected, `sp/3` is linear in the number of edges, regardless of whether a lattice or partial order is used for answer subsumption. The space required is virtually the same for both approaches, and the times are also quite similar, indicating that the worst-case complexity of partial order answer subsumption (Section 2) is not a factor for these examples.

Tests of `reach/3` on the same graphs show a similar growth in times to `sp/3` and virtually the same space. `reach/3` is about 3-4 times faster, indicating the overhead for answer subsumption on this simple example; it should be noted that shortest path uses answer subsumption extremely heavily, and the overhead for answer subsumption on most other programs will be much smaller. Profiling of `sp/3` shows that no deletions are performed on either the sparse-graph or dense-graph benchmarks. In these experiments, shorter paths are discovered first; when non-optimal paths are derived later, so that execution of answer subsumption code fails on the comparison in line 10 of Figure 2, and a deletion need not be performed. To test the cost of deletions, `sp_del/3` was tested on graphs where each edge fact also contains a randomly-generated cost.[5]

[5] The graphs used for `sp_del/3` have different randomly-generate edge relations than those for `sp/3` and so have a different number of answers.

Avg. Verts//Node	2	8	32	128	512
sp/3-Lattice					
Time	2.3	13.2	52.1	211.9	880
Table Space	26,249,826	41,213,664	41,213,664	41,213,664	41,213,664
Answers	631,509	1,000,000	1,000,000	1,000,000	1,000,000
sp/3-PO					
Time	4.1	16.5	56	218.2	890
Table Space	26,249,860	41,213,688	41,214,084	41,214,084	41,214,084
reach/3					
Time	0.88	3.47	12.5	53.2	238
Table Space	26,241,796	41,205,624	41,205,624	41,205,624	41,205,624
sp_del/3					
Time	4.2	104.0	329	845	2392
Table Space	27,198,048	41,203,908	41,290,552	41,322,464	41,345,080
Answers	655,221	999,000	1,000,000	1,000,000	1,000,000
Deletes	281,834	2,416,658	4,917,751	6,960,565	8,407,883

Table 2. Comparison of approaches on dense graphs where $|edges| = N \times |vertices|$

Table 2 shows that deletion imposes overhead in terms of time, but virtually no overhead in terms of space.

Comparison of answer subsumption to negation. In addition to using answer subsumption, shortest paths can also be computed through negation, as by the predicate pref_distance/4 in Figure 6, which concludes a given path between two vertices is shortest if no other shorter path is derivable. This approach is similar to a preference-

```
:- table pref_distance/4.
pref_distance(X,Y,1,_):- edge(X,Y).
pref_distance(X,Z,N,Max):-
   pref_distance(X,Y,N1,Max),
   edge(Y,Z), N is N1 + 1, N < Max,
   tnot(preferred_distance(X,Z,N,Max)).   % XSB's tabled negation

:- table preferred_distance/4.
preferred_distance(X,Y,N,Max):- pref_distance(X,Y,M,Max),M < N.
```

Fig. 6. A Program to Compute Shortest Path using Negation

based approach, where a shorter path is prefered to a longer one. Note that when answer subsumption is not used for shortest path, a program may have an infinite model if the underlying graph has cycles. To ensure termination, pref_distance/4 has as its fourth argument the maximum diameter of a graph. The need for a maximum distance, together with the requirement that calls to negative literals be ground, increases the complexity of determining shortest path. Not surprisingly, experiments show that pref_distance/4 scales poorly compared to the approaches based on answer subsumption. Since many ASP grounders may require users to program shortest path in a

ground manner similar to that of `pref_distance/3`, experiments on ASP grounders were also performed. The experiments showed poor scalability compared to answer subsumption. Overall, these results indicate that answer subsumption can play an important role for ASP grounding, either by implementing answer subsumption within a grounder, or by using TLP as a grounder as in XSB's XASP package.

4.2 Answer Subsumption and Abstract Interpretation

Net-style formalisms, such as Petri Nets, Workflow Nets, etc. have been used extensively for process modeling. Reachability is a central problem in analyzing properties of such nets, to which properties such as liveness, deadlock-freedom, and the existence of home states can be reduced. However, many interesting net formalisms cannot guarantee a finite number of configurations in a given net, so abstraction methods must be applied for their analysis.

For instance, the lack of finiteness is a problem in analyzing Place/Transition (PT) Nets. PT nets have no guard conditions or after-effects, and do not distinguish between token types. However, PT nets do allow a place to hold more than one token, leading to a potentially infinite number of configurations. This can be seen in the simple network of Figure 7 (from [3]) in which transitions are denoted by squares and places by circles. Each transition removes one token from the places that are the sources of its input edges and adds one token to each place at the target of each of its output edges. Starting from the configuration in Figure 7, repeated application of transition `t1` leads to place `s2` containing an unbounded number of tokens; repeated application of the sequence `t1,t2,t3,t4` leads to place `s4` containing an unbounded number of tokens.

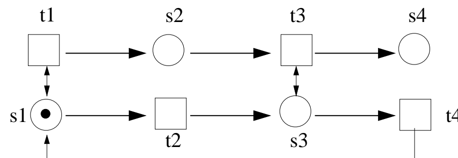

Fig. 7. A PT-net and configuration with an infinite number of reachable configurations

Despite such examples, reachability in PT nets is decidable and can be determined using an abstraction method called ω-sequences, (see e.g. [3]). The main idea in determining ω sequences is to define a partial order $\geq_\omega$ on configurations as follows. If configurations C_1 and C_2 are both reachable, C_1 and C_2 have tokens in the same set PL of places, C_1 has at least as many tokens in each place as C_2, and there exists a non-empty $PL_{sub} \subseteq PL$, such that for each $pl \in Pl_{sub}$ C_1 has strictly more tokens than C_2, then $C_1 >_\omega C_2$. When evaluating reachability, if C_2 is reached first, and then C_1 was subsequently reached, C_1 is abstracted by marking each place in PL_{sub} with the special token ω which is taken to be greater than any integer. If C_1 was reached first and then C_2, C_2 is treated as having already been seen.

Tabling combined with partial order answer subsumption requires slightly over 100 lines of code to model reachability in PT nets using ω-sequences. Due to space re-

strictions, the program cannot be fully described here, but the top-level reachability predicate is shown in Figure 8. Despite its succinctness, it can evaluate reachability in networks with millions of states in a few minutes. This use of tabling to determine reachability in PT nets can be seen as a special case of tabling for abstract interpretation (cf. [5] and other works). However the framework for answer subsumption described here allows tabling to be used to efficiently perform abstract interpretation within a general Prolog system

```
:- table reachable(_,po(omega_gte/2,omega_abs/3)).
reachable(InConf,NewConf):-
        reachable(InConf,NewConf),
        hasTransition(Conf,NewConf).
reachable(InConf,NewConf):- hasTransition(InConf,NewConf).
```

Fig. 8. Top-level predicate for PT net reachability

4.3 Scalability for multi-valued and quantitative logics

The technique of program justification (cf. e.g. [7]) has been used for debugging tabled programs that cannot be debugged by traditional means. Here, we consider justification in the context of the Silk system, currently under development at Vulcan, Inc. Silk is a commercial knowledge representation and rule system built on top of Flora-2, which is implemented using XSB. One of the salient features of Silk is its default reasoning, which is based on a parameterized argumentation theory evaluated under the well-founded semantics [12]. One issue in using Silk is that knowledge engineers must have a way of understanding the reasoning of the system, a task complicated by the use of the well-founded semantics and the intricacies of the argumentation theory. We describe an experimental approach to justification of Silk-style argumentation theories using multi-valued logics.

As noted in [12], argumentation theories in Silk are usually extensions of the default theories of Courteous Logic Programs (CLP) and are based on two user-defined predicates: `opposes/2` and `overrides/2`. Two atoms *oppose* each other if no model of a program can contain both atoms: an atom and its explicit negation oppose each other, but opposition can capture many other types of contradictions. Given two opposing atoms, one atom may *override* the other, and so be given preference. For atoms A_1 and A_2, if A_1 and A_2 are both derivable and oppose each other but neither overrides the other, A_1 and A_2 mutually *rebut* each other. If in addition A_1, say, overrides A_2, A_1 *refutes* A_2 [6]. Within Silk and Flora-2, the compilation of an argumentation theory ensures that rebutted atoms have an undefined truth value, as do atoms that refute themselves (i.e. if the `overrides/2` predicate is cyclic). However, for justification, it is meaningful to distinguish those facts that are undefined due to a negative loop in the argumentation theory from those that are undefined due to a negative loop in the program itself. In addition, it is meaningful to distinguish an atom that is true because

[6] In [12] argumentation theories are built on named rules, here we base them on derived atoms.

it overrides some other atom, from an atom whose derivation does not depend on the argumentation theory. Similar distinctions can be made for default false literals leading to the truth lattice shown in Figure 9.

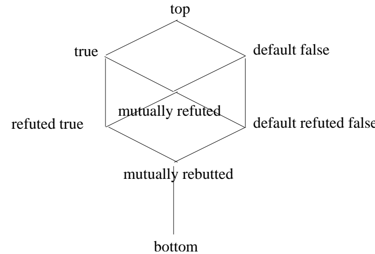

Fig. 9. A Truth Lattice for a Simplified Version of Courteous Argumentation Theory

An atom-based argumentation theory is added to a program by an easy *standard* transformation [12]. Each clause H :- B whose head is a defeasable predicate is rewritten as H :- $B, tnot(defeated(H))$; clauses for non-defeasible predicates are not altered. To obtain support for a justification a *multi-valued* transformation was used instead of the standard transformation. First, the lattice of Figure 9 was programmed in Prolog for use by answer subsumption. Next, each clause H :- B whose head was a defeasible predicate was rewritten as H :- $B, defeated(H, Reason)$, where `defeated(H,Value)` indicates the truth value of H on the lattice of Figure 9.

Experiments were performed on synthetic programs to compare the implementation of a Silk argumentation theory using the standard transformation to the new multi-valued transformation. Synthetic programs were tested containing a large number of mutually recursive defeasible rules, together with a large proportion of refuted and rebutted atoms. These tests indicate that the use of lattices may increase the time for total query evaluation by two to three times, well within an allowable increase for a justification system. Surprisingly, the multi-valued transformation of the argumentation theory sometimes take *less* table space, due to the space overhead incurred by XSB to maintain conditional answers (i.e. answers whose truth value is undefined in the well-founded semantics). We stress that these results are preliminary in the sense that the behavior of the synthetic programs may not resemble that of practical programs that use defeasible logic. However, the heavy use of defeasibility in the synthetic programs gives reason to believe that the time overhead may well be much less in practical programs than observed here. Together the results show that multi-valued logics are a promising approach for justification of defeasible logics, whether these logics occur as part of Silk or are used directly in a TLP system such as XSB.

5 Conclusions

This paper has described how answer subsumption can be used for applications in quantitative reasoning, abstract interpretation and multi-valued logics. To use answer subsumption, a programmer need only write a join, comparison, or abstraction operation

in Prolog and make the appropriate declarations. As shown in Section 3, the main implementational requirements of answer subsumption are 1) an efficient way to compare a new answer to appropriate answers in a table; and 2) an efficient way to delete subsumed answers. These features only access table space, so that they can be implemented by any tabling system, regardless of the engine architecture. Since XSB's table space is trie-based, other Prologs with trie-based tabling such as YAP or Ciao may be able to port XSB's engine code directly [7].

Answer subsumption is restricted to stratified programs in the current version of XSB. Future work includes the ability to use answer subsumption in non-stratified programs, and to add program constructs that allow non-idempotent aggregate operations to be computed, such as sum and count. However, the main work will be incorporating answer subsumption in applications such as program analysis in compilers, grounders for ASP solvers, and para-consistent and quantitative programs.

Acknowledgements The authors would like to thank Prasad Rao who helped implement the original version of answer subsumption, and Neng-Fa Zhou for a helpful discussion of tabling declarations.

References

1. C. V. Damásio and L. M. Pereira. Monotonic and residuated logic programs. In *ECSQARU*, pages 748–759, 2001.
2. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programs. In *IJCAI*, 2003.
3. J. Desel and W. Reisig. Place/transition Petri nets. In *Lectures on Petri Nets I: Basic Models*, pages 122–174. Springer LNCS 1491, 1998.
4. E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269 V27, 1959.
5. T. Kanamori and T. Kawamura. Abstract interpretation based on OLDT resolution. *JLP*, 15:1–30, 1993.
6. M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *JLP*, 12(4):335–368, 1992.
7. G. Pemmasani, H. Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *FLOPS*, pages 24–38, 2004.
8. F. Riguzzi and T. Swift. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *ICLP*, 2010. To appear.
9. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM TOPLAS*, 20(3):586 – 635, May 1998.
10. T. Swift. Tabling for non-monotonic programming. *AMAI*, 25(3-4):201–240, 1999.
11. T. Valente and K. Fujimoto. Bridges: Locating critical connectors in a network. *Social Networks*, 2010. To Appear.
12. H. Wan, B. Grossof, M. Kifer, P. Fodor, and S. Liang. Logic programming with defaults and argumentation theories. In *ICLP*, pages 432–448, 2009.
13. S. Wasserman and K. Faust. *Social Network Analysis*. Cambridge University Press, 1994.

[7] A significant amount of low-level C code has been ported from XSB to YAP to support a different feature termed call subsumption.