Constraints: An International Journal, 2, 337–359 (1997)

Memoing Evaluation for Constraint Extensions of Datalog *

DAVID TOMAN david@cs.toronto.edu
Department of Computer Science, University of Toronto

Editor: Raghu Ramakrishnan and Peter Stuckey

Abstract. This paper proposes an efficient method for evaluation of deductive queries over constraint databases. The method is based on a combination of the top-down resolution with memoing and the closed form bottom-up evaluation. In this way the top-down evaluation is guaranteed to terminate for all queries for which the bottom-up evaluation terminates. The main advantage of the proposed method is the direct use of the information present in partially instantiated queries without the need for rewriting of the original program. The evaluation algorithm automatically propagates the necessary constraints during the computation. In addition, the top-down evaluation potentially allows the use of compilation techniques, developed for compilers of logic programming languages, which can make the query evaluation very efficient.

Keywords: Datalog, constraint class, top-down evaluation, memoing evaluation of logic programs, SLG.

1. Introduction

We propose a new method for evaluating deductive queries over constraint databases (Kanellakis et al., 1995). The evaluation of queries over such databases is different from the one used in standard database systems. The constraints are used as the actual representation of data stored in the database rather than mere restrictions of the contents of otherwise ground relations. Algorithms for query evaluation over constraint databases have to satisfy the following criteria:

1. the evaluation algorithm has to terminate for all input queries,
2. the algorithm should be able to encompass various classes of constraints over wide range of domains, and
3. partially instantiated queries have to be evaluated efficiently.

The first requirement is especially difficult to achieve in the case of constraint databases: the extents of constraint relations are often infinite. There are two main approaches to satisfy the above requirements in the case of Datalog. However, neither of them seems to address all three of the requirements.

The first approach is based on a fixpoint, bottom-up evaluation of the rules. Here the first condition is usually met, e.g., for Datalog (Ullman, 1989), Datalog with dense order constraints (Kanellakis et al., 1995), Datalog with integer constraints (Revesz, 1993, Toman et al., 1994), and sets (Srivastava et al., 1994). However, the evaluation process is not *goal-oriented* and thus the evaluation of partially instantiated queries is fairly inefficient. Application of standard program transformation techniques, e.g., the Magic Rewriting, does not completely solve the problem (cf. Section 4).

* A preliminary report on this work appeared in Proc. 1995 ILPS, Portland, OR, (Toman, 1995).

The second approach is based on a top-down, resolution-based method. Here the second and third conditions are usually met. However, the termination guarantees are often sacrificed (Ullman, 1989) in order to improve the expressiveness and efficiency; an exception is (Swift and Warren, 1994b) where no constraints are allowed. On the other hand these methods can take full advantage of compilation techniques developed for other logic programming languages, e.g., (Freire et al., 1996, Swift and Warren, 1994a, Warren, 1983). This greatly improves the practical efficiency of query evaluation in the case of Datalog (Swift and Warren, 1994b). We show that similar results can be achieved for constraint extensions of Datalog.

In this paper we try to combine the advantages of the above two approaches. We propose an evaluation method, *Constraint Memoing*, applicable to constraint-based extensions of Datalog (Datalog$^{\mathcal{C}}$), that has the following features:

- Integrated Constraint Representation. Constraint Memoing integrates the constraints as *first-class* data into the evaluation procedure. This approach is different from most CLP systems, where constraints are handled by a separate *constraint solver* (Jaffar and Maher, 1994). We propose much tighter integration of constraints into the query evaluation: they are handled very similarly to standard ground tuples (or terms in the CLP systems). This is achieved by defining several *constraint operations* over the representation of the constraints that are used by the query evaluation algorithm (cf. Definition 2). Moreover, the same operations are also needed for the bottom-up evaluation (Kanellakis et al., 1995) and thus we can reuse results obtained in (Kanellakis et al., 1995, Revesz, 1993, Srivastava et al., 1994, Toman et al., 1994).

- Termination. Constraint Memoing guarantees termination of queries for all classes of constraints that have a terminating closed-form bottom-up evaluation procedure. Also, the complexity bounds of the bottom-up procedure are preserved.

- The expressiveness of the language can be easily extended to accommodate various classes of constraints as long as every class of constraints is equipped with several elementary operations on the underlying representation of the constraints. This step is quite subtle if termination of queries is to be preserved. In contrast to bottom-up methods, it is also possible to extend the query language to classes of constraints, where termination is not guaranteed. Even in those cases the algorithm reduces the possibility of non-termination (Sagonas et al., 1994).

- The use of a top-down method allows a fully goal-oriented query evaluation: the information present in partially instantiated queries is used to prune the search space of queries. The efficiency achieved by this method is better than the efficiency of comparable bottom-up methods including program rewriting techniques (e.g., Magic Set Transformation).

- The top-down evaluation strategy allows a direct use of the results obtained in the area of compilation techniques for logic programming languages (Gao and Warren, 1993, Swift and Warren, 1994a, Warren, 1983). Handling the constraints as first-class data allows us to use these techniques for query evaluation in constraint databases.

In (Ullman, 1989) the bottom-up approach (equipped with a query transformation phase) is shown to be no worse than the top-down approach for restricted classes of Datalog programs over ground relations. We show that the top-down approach is no worse than the bottom-up approach in the worst case, and in many empirical examples the top-down evaluation is much faster than the bottom-up evaluation of the same query.

The rest of the paper is organized as follows: section 2 introduces the constraint representation, the abstract constraint operations, and a closed form bottom-up evaluation procedure for Datalog$^{\mathcal{C}}$ in terms of these operations. Section 3 describes the proposed evaluation method, Constraint Memoing, includes the soundness, completeness, and termination proofs, and discusses possible optimization techniques specific to the proposed method. Section 4 introduces a general Magic Templates transformation (MT$^{\mathcal{C}}$) for Datalog$^{\mathcal{C}}$ for comparison purposes. Section 5 studies both the analytical complexity of query evaluation using Constraint Memoing and gives results that provide empirical evidence of the practicality of the proposed evaluation method. Section 6 concludes the presentation with a brief discussion of the related work and with possibilities of further improvements and directions for research.

2. Preliminaries

This section introduces the basic building blocks in terms of which the evaluation of Datalog$^{\mathcal{C}}$ queries is defined. Also, for reference, the standard bottom-up query evaluation procedure is introduced in terms of these building blocks.

DEFINITION 1 *Let $\mathcal{C}_0$ be a set of satisfiable atomic constraints. We define $\mathcal{C}$ to be the least set of constraints closed under the following rules:*

1. true $\in \mathcal{C}$.
2. $\mathcal{C}_0 \subseteq \mathcal{C}$.
3. *if $C_1, C_2 \in \mathcal{C}$ and $C_1 \wedge C_2$ is satisfiable then $C_1 \wedge C_2 \in \mathcal{C}$.*
4. *if $C \in \mathcal{C}$ and $x \in FV(C)$ then there exists a quantifier free formula $C_1 \vee \ldots \vee C_k$ (in DNF) equivalent to $\exists x.C$ such that $C_i \in \mathcal{C}$ for every satisfiable C_i where $0 < i \leq k$.*
5. *if $C \in \mathcal{C}$ and θ is a renaming of variables then $C\theta \in \mathcal{C}$.*

$FV(C)$ denotes the set of free variables in C.

This definition is similar to the definition of Constraint Domain (Jaffar and Maher, 1994). However, $\mathcal{C}$ contains only satisfiable constraints. The elements of $\mathcal{C}$ are used as a finite representation of the (possibly infinite) relations stored in a constraint database. The query evaluation over such a representation is based on the following operations:

DEFINITION 2 (CONSTRAINT CLASS) *Let $\mathcal{V}$ be a set of variables. A* Constraint class *is a set of constraints $\mathcal{C}$ from Definition 1 equipped with the following (computable) operations:*

Constraint Conjunction $\wedge^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \to \mathcal{C} \cup \{\perp\}$ *that for every pair of constraints C_1, C_2 computes the* conjunction $C_1 \wedge C_2$ *if the conjunction is satisfiable; otherwise it fails (returns $\perp$).*

Constraint Projection $\exists^{\mathcal{C}} : \mathcal{P}_{\text{fin}}(\mathcal{V}) \times \mathcal{C} \rightarrow \mathcal{P}_{\text{fin}}(\mathcal{C})$ *that for every constraint C and every finite set of variables V computes the set $\{C_i\}$ that satisfies the condition*

$$\left(\bigvee_{C_i \in \exists^{\mathcal{C}}(V,C)} C_i \right) \equiv \exists x_1 . \cdots \exists x_l . C$$

where $x_j \in V$ for $0 < j \leq l$. Note that by Definition 1 the function is well defined and always returns a finite *subset of $\mathcal{C}$.*

Constraint Subsumption $\leq^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \rightarrow$ bool *that satisfies following condition:*

$$C_1 \leq^{\mathcal{C}} C_2 \textit{ implies } C_1 \supset C_2$$

The first two operations are, in the world of constraints, the equivalents of relational algebra join and projection operations. However, while in the case of ground tuples the projection returns always only one tuple (constraint), in the case of more general constraints the constraint projection may return a set containing more than one constraint representing a disjunction (Toman et al., 1994, Williams, 1976).

EXAMPLE: In (Toman et al., 1994) we considered the following example: Assume that we want to eliminate quantifier $\exists y$ from the constraint:

$$(\exists y)(x + c_1 < y \wedge y + c_2 < z \wedge y \equiv_k d)$$

Clearly we cannot replace it simply by $x + c_1 + c_2 + 1 < z$ as in the case of gap-order constraints: we need to take into account the periodicity constraint $y \equiv_k d$, i.e., we need to make sure that there is at least one integer of the form $\{d + nk\}$ between $x + c_1$ and $z - c_2$. Thus, the equivalent quantifier-free formula is

$$\begin{array}{lcl} (x + c_1 + 1) \equiv_k d & \wedge & x + c_1 + c_2 + 1 < z \quad \vee \\ (x + c_1 + 2) \equiv_k d & \wedge & x + c_1 + c_2 + 2 < z \quad \vee \\ & \vdots & \\ (x + c_1 + k) \equiv_k d & \wedge & x + c_1 + c_2 + k < z \end{array}$$

It is easy to see that the variable y was successfully eliminated and the resulting constraint is a disjunction of conjunctions of periodicity and gap-order constraints. □

The last operation, the constraint subsumption, replaces the duplicate elimination for ground tuples. Note that the $\not\leq^{\mathcal{C}}$ is not unique by definition and does not have to imply $\not\supset$. However, a *better approximation* of $\supset$ relation by the $\leq^{\mathcal{C}}$ operation reduces the number of possible duplicate answers and improves the efficiency of the evaluation methods. In the following text we omit the superscripts $^{\mathcal{C}}$. We also use a *strict* ($\perp$-preserving) version of $\wedge^{\mathcal{C}}$.

The following definition states a fundamental property of constraint classes on which the termination proofs of bottom-up query evaluation procedures are implicitly based (Revesz, 1993, Srivastava et al., 1994, Toman et al., 1994, Ullman, 1989).

DEFINITION 3 (CONSTRAINT-COMPACT CLASS OF CONSTRAINTS) *Let $\mathcal{C}$ be a constraint class. If for every finite set of variables V and for every set $\mathcal{D} \subseteq \mathcal{C}$ such that $\forall C \in \mathcal{D}.FV(C) \subseteq V$ there is a finite subset $\mathcal{D}_{\text{fin}} \subseteq \mathcal{D}$ such that $\forall C \in \mathcal{D}.\exists C' \in \mathcal{D}_{\text{fin}}.C \leq^{\mathcal{C}} C'$ (i.e., $\mathcal{D}_{\text{fin}}$ covers $\mathcal{D}$ with respect to $\leq^{\mathcal{C}}$) then $\mathcal{C}$ is* constraint-compact.

This property plays a central role in the termination proofs of both the bottom-up based query evaluation procedures (cf. Section 2.1) and the top-down query evaluation procedure developed in Section 3. In general, the above condition could be weakened to require only that every infinite set of constraints contains a finite cover (where every constraint is covered by possibly several elements of the cover). However, the use of the weaker definition may require much more expensive subsumption checks (Srivastava, 1993). The two definitions are equivalent for constraint classes that satisfy the *single subsumption property* (Maher, 1993).

Example 4 (Common Constraint Classes) Standard Datalog can be defined using the class of constraints generated from the set $\{x = a : a \in A\}$ where A is the set of all constants in the Datalog program (the *active domain* (Abiteboul et al., 1995)).

Allowing general equality may cause problems to the standard evaluation strategies (rules may not be range-restricted). However, in our case we simply generate the appropriate class of constraints from the set $\{x = a : a \in A\} \cup \{x = y\}$. The evaluation remains otherwise unchanged as we use more general evaluation mechanism.

Incorporation of more interesting constraints, e.g., constraints over integers (Z) is also easy: the *gap-order* constraints (Revesz, 1993) are generated from the set $\mathcal{C}_{<Z} = \{x < u : u \in A\} \cup \{u < x : u \in A\} \cup \{x + c < y : c \in Z^+\}$. Similarly the *periodicity* constraints are generated from $\mathcal{C}_{\equiv Z} = \{x \equiv_k c : c \in A\}$. In (Toman et al., 1994) a closed form bottom-up evaluation procedure for $\mathcal{C}_{\equiv,<Z} = \mathcal{C}_{<Z} \cup \mathcal{C}_{\equiv Z}$ was developed including the constraint operations from Definition 2. The *dense order* constraints over Q can be incorporated by a slight modification of constraint operations defined in (Kanellakis et al., 1995).

All the above constraint classes are constraint-compact. However, there are also constraint classes where all the constraint operations are defined, but which are *not* constraint-compact, e.g., the class generated from the set $\{x + c < y : c \in Z\}$ (gap-order constraints with possibly negative size of the gap (Revesz, 1993)) or the linear arithmetic constraints (Kanellakis et al., 1995).

DEFINITION 5 (DATALOG$^{\mathcal{C}}$ PROGRAM) *Let $\mathcal{C}$ be a class of constraints. A* atom *is a predicate symbol with distinct variables as its arguments. A* Datalog$^{\mathcal{C}}$ *is a set of clauses of the form*

$$A \leftarrow D, B_1, \ldots, B_k$$

where A and B_i are atoms and $D \in \mathcal{C}$.

We assume that the extensional database (EDB) is represented by a set of unit clauses $A \leftarrow D$ as a part of the Datalog$^{\mathcal{C}}$ program. A query over such database is a tuple containing an atom and a constraint the returned tuples have to satisfy.

DEFINITION 6 (QUERY) *Let P be a Datalog$^{\mathcal{C}}$ program, G an atom, and $C \in \mathcal{C}$. We call the tuple (G, C, P) a* query. *The* answer *to the query (G, C, P) is a set of valuations θ such that $P \models (G \wedge C)\theta$. A* query evaluation procedure *is an algorithm that computes an answer to the query. A query is* partially instantiated *if the constraint C is nontrivial (i.e., $C \neq$* true*).*

2.1. Closed-form Bottom-up Evaluation

The usual approach to query evaluation for Datalog$^{\mathcal{C}}$ is a variation on the bottom-up evaluation algorithm (Ullman, 1989). In its simplest form a bottom-up evaluation algorithm is defined as follows:

DEFINITION 7 (INTERPRETATION) *Let $R(x_1, \ldots, x_k)$ be an atom and $C \in \mathcal{C}$ a constraint such that $FV(C) \subseteq FV(R)$. A pair (R, C) is a* constraint atom. *A ($\mathcal{C}$-)*interpretation *is a set of constraint atoms.*

Constraint atoms play the role of ground atoms (tuples) stored in a standard relational system. The definition of the TP operator is now similar to the definition of the corresponding operator on ground atoms. However, in this case all the operations in the definition of TP are defined with respect to the chosen class of constraints $\mathcal{C}$ (see Definition 2).

DEFINITION 8 (IMMEDIATE CONSEQUENCE OPERATOR) *Let P be a Datalog$^{\mathcal{C}}$ program and I a $\mathcal{C}$-interpretation. We define*

$$\begin{aligned}\mathrm{TP}_{\mathcal{C}}(I) = \{(A, C') : \; & A \leftarrow D, B_1, \ldots, B_k \in P, \\ & (B_i, C_i) \in I \textit{ for all } 0 < i \leq k, D \in \mathcal{C}, \\ & C = D \wedge C_1 \wedge \ldots \wedge C_k \textit{ exists, } C' \in \exists_A.C, \\ & \textit{and if } C' \leq C'' \textit{ for some } (A, C'') \in I \textit{ then } C' = C''\}\end{aligned}$$

where $\exists_A$ is a shorthand for $\lambda x.\exists^{\mathcal{C}}(FV(x) - FV(A), x)$. The variables in the constraints are renamed using the variable names in the associated atoms of the clause.

The bottom-up evaluation algorithm remains unchanged: all the modifications needed for the evaluation of *constraint* queries are encapsulated in the definition of the $\mathrm{TP}_{\mathcal{C}}$ operator.

Algorithm 9 (Naive Bottom-up Evaluation) *Let (G, C, P) be a query. The following algorithm computes the answer to this query.*

> $I := \emptyset$
> *repeat*
> $J, I := I, \mathrm{TP}_{\mathcal{C}}(I)$
> *while* $J \neq I$
> *return* $\{C \wedge D : (G, D) \in I\}$

This arrangement also shows how other TP-based evaluation procedures can be utilized for constraint query evaluation, e.g., the semi-naive bottom-up evaluation (Ullman, 1989). Algorithm 9 was shown to be sound and complete for Datalog (Ullman, 1989), Datalog$^{<Q}$, and Datalog$^{<,\equiv Z}$ (Revesz, 1993, Toman et al., 1994). A simple generalization of the proofs in (Toman et al., 1994) shows soundness and completeness of Algorithm 9 for a general class $\mathcal{C}$:

Notation 10 *Let $S \subseteq \mathcal{C}$. Then $||S||$ is the set of valuations θ such that $\theta \models C$ for some $C \in S$. For a constraint interpretation I we define $||I|| = \{A\theta : (A, C) \in I, \theta \models C\}$.*

Theorem 11 (Soundness and Completeness) *Let P be a Datalog$^{\mathcal{C}}$ program. Then*

$$||\,\mathrm{TP}_{\mathcal{C}}^{\omega}(\emptyset)|| = \mathrm{TP}^{\omega}(\emptyset).$$

Proof: *By simultaneous induction on stages of* TP *and* $\mathrm{TP}_{\mathcal{C}}$. *The base case holds vacuously. Let* $i > 0$. $G\vartheta \in \mathrm{TP}^i(\emptyset)$. *Then there is* θ *an extension of* ϑ *and a clause* $G \leftarrow D, B_1, \ldots, B_k$ *in* P *such that* $\theta \models D$ *and* $B_i\theta \in \mathrm{TP}^{i-1}(\emptyset)$. *Then by the induction hypothesis* $B_j\theta \in \|\mathrm{TP}_{\mathcal{C}}^{i-1}(\emptyset)\|$. *Thus for each* B_j *there is a constraint* C_j *such that* $(B_j, C_j) \in \mathrm{TP}_{\mathcal{C}}^{i-1}(\emptyset)$ *and* $\theta \models D \wedge C_1 \wedge \ldots \wedge C_k$. *By definition of* $\mathrm{TP}_{\mathcal{C}}$ *there is* $A \in \exists_G.D \wedge C_1 \wedge \ldots \wedge C_k$ *such that* $\vartheta \models A$. *Therefore* $\vartheta \in \|\mathrm{TP}_{\mathcal{C}}^i(\emptyset)\|$. *The other direction is similar.* ■

Theorem 12 (Termination) *Let* $\mathcal{C}$ *be a constraint-compact class of constraints. Then Algorithm 9 terminates for every Datalog*$^{\mathcal{C}}$ *query.*
Proof: *Immediate from Definitions 3 and 8. Assume, that Algorithm 9 does not terminate. Then in every iteration it generates a constraint atom that is not subsumed by any previously generated constraint atom. As there are only finitely many different predicate symbols in every Datalog*$^{\mathcal{C}}$ *program, there must be at least one symbol, that occurs infinitely often among the generated atoms. However, this is an infinite set of constraints over a fixed finite set of variables and thus it must contain a finite constraint cover by Definition 3; a contradiction.* ■

All the constraint classes in Example 4 have a closed-form terminating bottom-up evaluation procedure (based on Definitions 8 and 9).

2.2. *Goal-oriented Evaluation Strategies*

There are several standard improvements to the naive bottom-up evaluation algorithm, e.g., the semi-naive algorithm (Ullman, 1989). However, these strategies fail to take into account the information contained in a partially instantiated query: they are not goal-oriented. There are two major approaches to solving this problem in the case of standard (ground) Datalog:

1. Rewrite the original program using the Magic Templates (MT) transformation technique (Bancilhon et al., 1986, Ramakrishnan, 1991) and subsequently evaluate the transformed program bottom-up, or

2. Adopt a variant of a top-down evaluation strategy (Clocksin and Mellish, 1987) based on the resolution principle (Lloyd, 1987).

In this paper we consider mainly the top-down, resolution-based methods. However, the MT optimization for constraint deductive query languages is also be introduced for comparison purposes. It is well known that the standard top-down strategies, e.g., the SLD-resolution (Clocksin and Mellish, 1987), despite their efficiency, have a major drawback as query evaluation procedures: they lead to nonterminating computations even in the situations, where the bottom-up algorithms are guaranteed to terminate. Note also that breadth-first traversal of a SLD-tree does not guarantee termination in general.

The drawback is caused by occurrences of infinite paths in SLD search trees. This has been observed in several papers, e.g., (Chen and Warren, 1993, Tamaki and Sato, 1986) and an alternative to SLD-resolution was proposed (under various names). The main idea lies in *remembering* answers for already resolved subgoals. This approach guarantees termination in the case of function-free logic programs (Swift and Warren, 1994b). We extend this method to *constraint* deductive queries while preserving the termination and complexity bounds of the bottom-up evaluation algorithms.

3. Top-down evaluation for Datalog$^{\mathcal{C}}$

In the last section the bottom-up evaluation of Datalog$^{\mathcal{C}}$ was defined in terms of elementary operations over a given constraint class. This section shows how a top-down query evaluation procedure (SLG-resolution (Chen and Warren, 1993)) can be refined using the same operations to handle constraint queries. This approach allows us to build an efficient top-down evaluation procedure for every class of constraints that has a closed form bottom-up evaluation procedure. Moreover, the termination and complexity bounds of the bottom-up algorithm are preserved.

The modification of the (positive fragment of the) SLG resolution for Constraint Memoing (SLG$^{\mathcal{C}}$ resolution) is defined by the following set of rewriting rules:

DEFINITION 13 (SLG$^{\mathcal{C}}$ REWRITING RULES) *Let $\mathcal{C}$ be a constraint class and*

$$\mathsf{root}(G;C), \mathsf{body}(G;B_1,\ldots,B_k;C), \mathsf{goal}(G;B,C';B_2,\ldots,B_k;C), \textit{ and } \mathsf{ans}(G,A)$$

where G is an atom, $B_1,\ldots,B_k$ are literals, and C, C', A are constraints in $\mathcal{C}$, be nodes from which we build SLG-trees using the following rules:

Action/Node	Children	Conditions
Clause resolution		
$\mathsf{root}(G;C)$	$\left\{\begin{array}{l}\mathsf{body}(G;B^1_1,\ldots,B^1_{k_1};C\wedge D^1)\\ \vdots\\ \mathsf{body}(G;B^l_1,\ldots,B^l_{k_l};C\wedge D^l)\end{array}\right.$	for all $0 < i \le l$ such that $G \leftarrow D^i, B^i_1,\ldots,B^i_{k_i} \in P$ and $C \wedge D^i$ is satisfiable
Query projection		
$\mathsf{body}(G;B_1,\ldots,B_k;C)$	$\left\{\begin{array}{l}\mathsf{goal}(G;B_1,C_1;B_2,\ldots,B_k;C)\\ \vdots\\ \mathsf{goal}(G;B_1,C_l;B_2,\ldots,B_k;C)\end{array}\right.$	for all $C_i \in \exists_{B_1}.C$
Answer propagation		
$\mathsf{goal}(G;B,C';B_2,\ldots,B_k;C)$	$\left\{\begin{array}{l}\mathsf{body}(G;B_2,\ldots,B_k;C\wedge A_1)\\ \vdots\\ \mathsf{body}(G;B_2,\ldots,B_k;C\wedge A_l)\end{array}\right.$	for all $A_i \in \mathrm{ans}(B,C'')$ where $C' \le_{\mathcal{C}} C''$ and $C \wedge A_i$ is satisfiable
Answer projection		
$\mathsf{body}(G;\ ;C)$	$\left\{\begin{array}{l}\mathsf{ans}(G;A_1)\\ \vdots\\ \mathsf{ans}(G;A_l)\end{array}\right.$	for all $A_i \in \exists_G.C$

where G, B_i, and B^j_i are atoms, $A_i, C, C', C'', C_i, D^i \in \mathcal{C}$, and $\mathrm{ans}(B,C'')$ is the set of answers collected from the leaves of the SLG-tree rooted by (B,C'') (introduced in Notation 16).

A SLG-tree *is a tree built from a node $\mathsf{root}(G;C)$ by a finite application of the above rules. A* SLG-forest *is a set of SLG-trees.*

Note that the *Answer propagation* and *Answer projection* rules have to *cooperate*: every time a new answer—an answer not subsumed by any answer in the same tree generated earlier—is produced, it is propagated to all the nodes that have already been resolved using answers from this particular tree. Also, the *Answer propagation* rule is responsible for creating new SLG-trees in the SLG-forest when no tree with a root node that subsumes the goal to be resolved can be found.

In the presented form the *Query projection* rule implements the left-to-right selection rule (common to most of the LP systems). However, any other goal selection strategy can be implemented here, e.g., selection based on specific SIPS—Sideways Information Passing Style (Ramakrishnan, 1991).

The main difference between SLG and SLG$^{\mathcal{C}}$ is in two additional rules: the *Query Projection* and the *Answer Projection*. The *Query Projection* rule is responsible for determining what are the goals to be resolved by answer resolution (i.e., what is the goal-constraint pair to be looked for among the already computed answers). The *Answer Projection* is responsible for storing the computed answers for a given goal for subsequent lookup and propagating them to the appropriate goal nodes. Note the essential use of constraint projection which allows to determine the relevant constraint for every atom.

The SLG-resolution can handle negation using additional rules (Swift and Warren, 1994b, Toman, 1997). However, our proposal currently allows only positive programs. Adding negation is briefly discussed in Section 6.

The SLG$^{\mathcal{C}}$ rewriting rules are used for the query evaluation as follows:

DEFINITION 14 *Let* $\mathrm{SLG}(G, C)$ *be the SLG-forest generated from the query* (G, C, P) *as follows:*

1. *create a SLG-forest containing a single tree* $\{\mathsf{root}(G; C)\}$.
2. *expand the leftmost node using the rules in Definition 13 as long as they can be applied.*
3. *return the set* $\mathrm{ans}(G, C)$ *as the answer for the query.*

Let $\mathrm{slg}(G, C)$ *be a SLG-tree rooted by the node* $\mathsf{root}(G; C)$.

We order the answers (i.e., the nodes $\mathsf{ans}(G; C)$) in the SLG-forest according to the order in time in which they were derived:

DEFINITION 15 *Let* $\mathsf{ans}(G; A)$ *and* $\mathsf{ans}(G'; A')$ *be leaves in the SLG-forest generated by the SLG-rules (cf. Definition 13). Then we say that* $\mathsf{ans}(G; A)$ is older than $\mathsf{ans}(G'; A')$ *if the node* $\mathsf{ans}(G; A)$ *is generated before the node* $\mathsf{ans}(G'; A')$.

We assume that the SLG-trees are ordered left-to-right in the order they were created. The choice of selection strategy does not affect the soundness/completeness of the algorithm (Lloyd, 1987). However, the selection strategy may influence the efficiency of the system (cf. Section 6).

Notation 16 *Let* $\mathrm{ans}(G, C)$ *be the set of all* A *such that* $\mathsf{ans}(G; A) \in \mathrm{slg}(G, C)$ *and if an older* $\mathsf{ans}(G; A') \in \mathrm{slg}(G, C)$ *then* $A \not\leq A'$.

Lemma 17 *Let* $\mathsf{ans}(G; A)$ *be a leaf of the SLG-tree* $\mathrm{slg}(G, C)$. *Then for every application of the answer propagation rule along the path* $\mathsf{root}(G; C) \to \ldots \to \mathsf{ans}(G; A)$ *if* $\mathsf{ans}(G'; A')$ *was a propagated answer then* $\mathsf{ans}(G'; A')$ *is older than* $\mathsf{ans}(G; A)$.

Proof: *Immediate from the Definitions 13, 14, and 15.* ■

Soundness and completeness of Constraint Memoing is proven by reduction to soundness and completeness of bottom-up evaluation (Algorithm 9). Note that the set $\mathrm{ans}(G, C)$ may not be unique and depends on the order in which the nodes $\mathsf{ans}(G; A)$ are generated. However, for our purposes it is sufficient that the set of valuations $\|\,\mathrm{ans}(G, C)\|$ is unique.

Lemma 18 *Let* $\mathrm{slg}(G, C)$ *be a SLG-tree and* ϑ *a valuation. Then* $\vartheta \in \|\,\mathrm{ans}(G, C)\|$ *implies* $\vartheta \models C$.
Proof: *By induction on the height of the SLG-tree* $\mathrm{slg}(G, C)$. ■

To prove correctness of the algorithm we show that all the derived answers are also derived in the bottom-up computation:

Lemma 19 *Let* (G_0, C_0, P) *be a query. Then for every* $\mathrm{slg}(G, C) \in \mathrm{SLG}(G_0, C_0)$ *and every valuation* ϑ

$$\vartheta \in \|\,\mathrm{ans}(G, C)\| \implies G\vartheta \in \mathrm{TP}^{\omega}(\emptyset).$$

Proof: *Induction on the "age" of answers: Let* $\mathsf{ans}(G; C') \in \mathrm{slg}(G, C)$ *such that* $\vartheta \models C'$. *Then there is a path*

$$\mathsf{root}(G; C) \to \mathsf{body}(G; B_1, \ldots, B_k; C_1) \to \ldots \to \mathsf{ans}(G; C') \in \mathrm{slg}(G, C)$$

that starts with a Clause Resolution *step using a clause* $G \leftarrow D, B_1, \ldots, B_k \in P$ *(cf. Figure 1). By Definition 13,* $C' \in \exists_G.C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$ *where* A_i *is an answer propagated from the SLG-trees* $\mathrm{slg}(B_i, C_i')$. *Thus, there exists* θ *an extension of* ϑ, *such that* $\theta \models C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$. *Clearly,* $\theta \models A_i$ *for* $0 < i \leq k$. *By Lemma 17 all the answers* $\mathsf{ans}(B_i; A_i)$ *used along this path have been computed before* $\mathsf{ans}(G; C')$*and thus by the induction hypothesis we have* $B_i\theta \in \mathrm{TP}^{\omega}(\emptyset)$. *By definition of* TP *and the fact that* $\theta \models D$ *we have* $G\theta = G\vartheta \in \mathrm{TP}^{\omega}(\emptyset)$. ■

Thus, all answers—not only for the main query, but also for all subqueries represented by the SLG-trees in the SLG forest generated from the main query—are sound.

Lemma 20 *Let* G *be an atom and* $C_1, C_2 \in \mathcal{C}$ *constraints. Then*

$$C_1 \leq C_2 \implies \|\,\mathrm{ans}(G, C_1)\| \subseteq \|\,\mathrm{ans}(G, C_2)\|.$$

Proof: *Immediate from the definition of* ans *and Lemma 18.* ■

The next thing to show is that the algorithm computes all the answers to the given query. This is a little bit more complicated, as the algorithm does not compute all the answers to an *uninstantiated* query like the bottom-up evaluation does. However we can show:

Lemma 21 *Let* (G_0, C_0, P) *be a query. Then for every* $\mathrm{slg}(G, C) \in \mathrm{SLG}(G_0, C_0)$ *and every valuation* ϑ

$$G\vartheta \in \mathrm{TP}^{\omega}(\emptyset) \wedge \vartheta \models C \implies \vartheta \in \|\,\mathrm{ans}(G, C)\|.$$

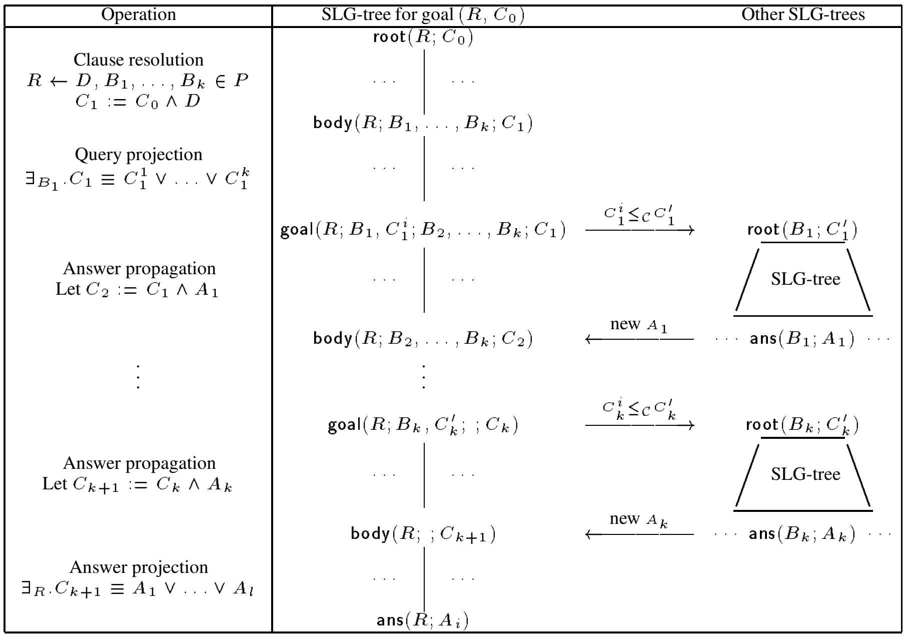


Figure 1. SLG^C Evaluation of goal R w.r.t. a constraint C_0.

Proof: *By induction on stages of* TP. *Let* $G\vartheta \in \mathrm{TP}^i(\emptyset)$ *and* $\vartheta \models C$. *The claim holds vacuously for* $i = 0$. *Let* $i > 0$. *Then there is a clause* $G \leftarrow D, B_1, \ldots, B_k \in P$ *and an extension* θ *of* ϑ *such that* $\theta \models D$ *and* $B_i\theta \in \mathrm{TP}^{i-1}(\emptyset)$. *We can construct a path in* $\mathrm{slg}(G, C)$ *that ends with a node* $\mathsf{ans}(G; A')$ *such that* $\vartheta \models A'$. *Using the assumption* $\vartheta \models C$ *we have* $\theta \models C \wedge D$ *(this corresponds to the application of the* Clause Resolution *rule). Thus,* $\theta \models C_1'$ *for at least one element* C_1' *of* $\exists_{B_1} C \wedge D$. *By the inductive hypothesis* $\theta \in \|\,\mathrm{ans}(B_1, C_1')\|$. *This fact in turn yields a node* $\mathsf{ans}(B_1; A_1) \in \mathrm{slg}(B_1, C_1')$ *such that* $\theta \models A_1$. *Again, using the previous assumptions,* $\theta \models C \wedge D \wedge A_1$. *In general, let* C_i' *be an element of* $\exists_{B_i}.C \wedge D \wedge A_1 \wedge \ldots \wedge A_{i-1}$. *Clearly, by* $i - 1$ *applications of the induction hypothesis,* $\theta \models C_i'$. *Then by induction hypothesis on* B_i *we have* $\theta \in \|\,\mathrm{ans}(B_i, C_i')\|$. *This exactly corresponds to an application of the* Query Projection *and* Answer Propagation *rules from Definition 13.*

After k steps we have $\theta \models C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$. *Thus,* $\vartheta \models A'$ *for some element* A' *of* $\exists_G.C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$ *(this is achieved by the* Answer Projection *rule), and therefore* $\vartheta \in \|\,\mathrm{ans}(G, C)\|$.

In the actual algorithm, the application of the Answer Propagation *rule does not necessarily use the tree* $\mathrm{slg}(B_i, C_i')$ *for answer resolution. However, if a different tree* $\mathrm{slg}(B_i, C_i'')$ *is used then it is always the case that* $C_i' \leq_C C_i''$. *Thus, by Lemma 20,* $\|\,\mathrm{ans}(B_i, C_i')\| \subseteq \|\,\mathrm{ans}(B_i, C_i'')\|$ *and no answers can possibly be lost.* ■

(1)	$\mathsf{root}(tc(X,Y);[X=b])$	initial query node
(2)	├ $\mathsf{body}(tc(X,Y);tc(X,Z),tc(Z,Y);[X=b])$	
(4)	└ $\mathsf{goal}(tc(X,Y);tc(X,Z),[X=b];tc(Z,Y);[X=b])$	get results for $X=b$
(8)	├ $\mathsf{body}(tc(X,Y);tc(Z,Y);[X=b,Z=c])$	answer from node (7)
(9)	└ $\mathsf{goal}(tc(X,Y);tc(Z,Y),[Z=c];;[X=b,Z=c])$	get results for $X=c$ $(\rightarrow 10)$
(18)	├ $\mathsf{body}(tc(X,Y);;[X=b,Z=c,Y=b]$	answer from node (17)
(20)	└ $\mathsf{ans}(tc(X,Y);[X=b,Y=b])$	new answer for $X=b$
(31)	├ $\mathsf{body}(tc(X,Y);;[X=b,Z=c,Y=c]$	answer from node (30)
(33)	└ $\mathsf{ans}(tc(X,Y);[X=b,Y=c])$	
(41)	└ $\mathsf{body}(tc(X,Y);;[X=b,Z=c,Y=d]$	answer from node (40)
(44)	└ $\mathsf{ans}(tc(X,Y);[X=b,Y=d])$	new answer for $X=b$
(21)	├ $\mathsf{body}(tc(X,Y);tc(Z,Y);[X=b,Z=b])$	answer from node (20)
(22)	└ $\mathsf{goal}(tc(X,Y);tc(Z,Y),[Z=b];;[X=b,Z=b])$	get results for $X=b$
(23)	├ $\mathsf{body}(tc(X,Y);;[X=b,Z=b,Y=c]$	answer from node (7)
(25)	└ $\mathsf{ans}(tc(X,Y);[X=b,Y=c])$	
(24)	├ $\mathsf{body}(tc(X,Y);;[X=b,Z=b,Y=b]$	answer from node (20)
(26)	└ $\mathsf{ans}(tc(X,Y);[X=b,Y=b])$	
(46)	└ $\mathsf{body}(tc(X,Y);;[X=b,Z=b,Y=d]$	answer from node (44)
(54)	└ $\mathsf{ans}(tc(X,Y);[X=b,Y=d])$	
(45)	└ $\mathsf{body}(tc(X,Y);tc(Z,Y);[X=b,Z=d])$	answer from node (44)
(48)	└ $\mathsf{goal}(tc(X,Y);tc(Z,Y),[Z=c];;[X=b,Z=d])$	get results for $X=d$ $(\rightarrow 49)$
(3)	└ $\mathsf{body}(tc(X,Y);e(X,Y);[X=b])$	
(5)	└ $\mathsf{goal}(tc(X,Y);e(X,Y),[X=b];;[X=b])$	
(6)	└ $\mathsf{body}(tc(X,Y);;[X=b,Y=c])$	
(7)	└ $\mathsf{ans}(tc(X,Y);[X=b,Y=c])$	new answer for $X=b$
(10)	$\mathsf{root}(tc(X,Y);[X=c])$	subquery from (9)
(11)	├ $\mathsf{body}(tc(X,Y);tc(X,Z),tc(Z,Y);[X=c])$	
(13)	└ $\mathsf{goal}(tc(X,Y);tc(X,Z),[X=c];tc(Z,Y);[X=c])$	get results for $X=c$
(19)	├ $\mathsf{body}(tc(X,Y);tc(Z,Y);[X=c,Z=b]$	answer from node (17)
(27)	└ $\mathsf{goal}(tc(X,Y);tc(Z,Y),[Z=b];;[X=c,Z=b]$	get results for $X=b$
(28)	├ $\mathsf{body}(tc(X,Y);[X=c,Z=b,Y=c]$	answer from node (7)
(30)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=c]$	new answer for $X=c$
(29)	├ $\mathsf{body}(tc(X,Y);[X=c,Z=b,Y=b]$	answer from node (20)
(34)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=b]$	
(47)	└ $\mathsf{body}(tc(X,Y);[X=c,Z=b,Y=d]$	answer from node (44)
(55)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=d]$	
(32)	├ $\mathsf{body}(tc(X,Y);tc(Z,Y);[X=c,Z=c]$	answer from node (30)
(35)	└ $\mathsf{goal}(tc(X,Y);tc(Z,Y),[Z=c];;[X=c,Z=c]$	get results for $X=c$
(36)	├ $\mathsf{body}(tc(X,Y);[X=c,Z=c,Y=b]$	answer from node (17)
(38)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=b]$	
(37)	├ $\mathsf{body}(tc(X,Y);[X=c,Z=c,Y=c]$	answer from node (30)
(39)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=c]$	
(42)	└ $\mathsf{body}(tc(X,Y);[X=c,Z=c,Y=c]$	answer from node (40)
(56)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=c]$	
(43)	└ $\mathsf{body}(tc(X,Y);tc(Z,Y);[X=c,Z=d]$	answer from node (40)
(57)	└ $\mathsf{goal}(tc(X,Y);tc(Z,Y),[Z=d];;[X=c,Z=b]$	
(12)	└ $\mathsf{body}(tc(X,Y);e(X,Y);[X=c])$	
(14)	└ $\mathsf{goal}(tc(X,Y);e(X,Y),[X=c];;[X=c])$	
(15)	├ $\mathsf{body}(tc(X,Y);;[X=c,Y=b])$	
(17)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=b])$	new answer for $X=c$
(16)	└ $\mathsf{body}(tc(X,Y);;[X=c,Y=d])$	
(40)	└ $\mathsf{ans}(tc(X,Y);[X=c,Y=d])$	new answer for $X=c$
(49)	$\mathsf{root}(tc(X,Y);[X=d])$	subquery from (48)
(50)	├ $\mathsf{body}(tc(X,Y);tc(X,Z),tc(Z,Y);[X=d])$	
(52)	└ $\mathsf{goal}(tc(X,Y);tc(X,Z),[X=d];tc(Z,Y);[X=d])$	
(51)	└ $\mathsf{body}(tc(X,Y);e(X,Y);[X=d])$	answer from node (40)
(53)	└ $\mathsf{goal}(tc(X,Y);e(X,Y),[X=d];;[X=d])$	

The First column indicates the order, in which the individual nodes have been created, the second column shows the SLG-trees generated, and the last column describes how answers have been generated by the SLG-trees; the relation e represents the edges in the graph $a \rightarrow b \rightleftarrows c \rightarrow d$.

Figure 2. Annotated SLG evaluation of query $tc(X,Y)$ for $X=b$.

By composing the previous Lemmas we have:

Theorem 22 (Soundness and Completeness) *Let (G, C, P) be a query tuple. Then for all valuations ϑ such that $\vartheta \models C$*

$$G\vartheta \in \mathrm{TP}^{\omega}(\emptyset) \iff \vartheta \in \|\,\mathrm{ans}(G, C)\|.$$

Proof: *Soundness follows from Lemmas 18 and 19, completeness from Lemma 21.* ■

The soundness and completeness proof is based on the reduction to the fixpoint computation on *ground* instances. However, to prove termination of the query evaluation algorithms (in both the bottom-up and top-down cases) a finite encoding of a potentially infinite result of the evaluation is needed (Revesz, 1993, Toman et al., 1994):

Theorem 23 (Termination) *Let $\mathcal{C}$ be a constraint-compact class. Then the SLG$^{\mathcal{C}}$ evaluation terminates for all queries (G, C, P).*
Proof: *Let $\mathcal{C}$ be a constraint-compact class of constraints. Then:*

1. *The number of trees in the SLG-forest $\mathrm{SLG}(G, C)$ is finite, as there are only finitely many predicate symbols $G' \in P$ and for every predicate symbol the set $\{C_i : \mathsf{root}(G', C_i) \in \mathrm{SLG}(G, C)\}$ is finite by Definitions 3 and 13.*
2. *Every* root *node has finitely many children, as there are only finitely many clauses in P.*
3. *Every* body *node has finitely many children, as the set $\exists_G.C$ is finite for any $C \in \mathcal{C}$.*
4. *Every* goal *node as only finitely many children, as there are only finitely many elements in the set $\mathrm{ans}(G', C')$ for any atom G' and $C' \in \mathcal{C}$ by Definition 3.*
5. *Every $\mathrm{slg}(G, C)$ has finite depth, because of finite number of subgoals in the bodies of each clause in P.*

Therefore the rules from Definition 13 can be applied only finitely many times. ■

The termination of the Constraint Memoing algorithm is guaranteed in all cases when the bottom-up algorithm terminates computing a finite interpretation $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$. Moreover, it is usually easy to *decompose* the original bottom-up evaluation procedure and extract the elementary operations on constraints needed for Constraint Memoing (Definition 2).

3.1. Optimization

To reduce the overhead introduced by the SLG$^{\mathcal{C}}$ resolution (in comparison to standard SLD resolution) we explore several possibilities:
1. Solving more general goals than necessary:

Action/Node	Children	Conditions
Query projection		
$\mathsf{body}(G; B_1, \ldots, B_k; C)$	$\left\{ \begin{array}{l} \mathsf{goal}(G; B_1, C_1; B_2, \ldots, B_k; C) \\ \vdots \\ \mathsf{goal}(G; B_1, C_l; B_2, \ldots, B_k; C) \end{array} \right.$	$\exists_{B_1}.C \supset C_1 \vee \ldots \vee C_l$ for some $C_1, \ldots, C_l$

This modification may reduce the number of SLG-trees in the SLG-forest (in cases where $|\exists_{B_1}.C| > l$). However, the propagation of constraints *at the time of goal resolution* is reduced. The soundness and completeness properties are preserved by Lemma 20. The termination is guaranteed similarly to Theorem 23. In (Gao and Warren, 1993) the following version of such a modification was considered:

Action/Node	Children	Conditions
Query projection		
$\mathsf{body}(G; B_1, \ldots, B_k; C) \to$	$\mathsf{goal}(G; B_1, \text{true}; B_2, \ldots, B_k; C)$	none

In this case, there is only one SLG-tree per predicate symbol. On the other hand, no constraints are propagated at the time of goal resolution—the constraints are used merely to restrict the returned answers. Thus, the computation essentially computes all answers to an *uninstantiated* query similarly to the bottom-up algorithm, and the performance suffers: The performance is approximately the same as evaluating the uninstantiated query.

2. Mixed SLG and SLD resolution (by memoing only subset of the predicate symbols present in the program).

Action/Node	Children	Conditions
Non-tabled resolution		
$\mathsf{body}(G; B_1, \ldots, B_k; C)$	$\begin{cases} \mathsf{body}(G; B_1^1, \ldots, B_{k_1}^1, \\ \qquad B_2, \ldots, B_k; C \wedge D^1) \\ \qquad \vdots \\ \mathsf{body}(G; B_1^l, \ldots, B_{k_l}^l, \\ \qquad B_2, \ldots, B_k; C \wedge D^l) \end{cases}$	for B_1 not tabled goal $B_1 \leftarrow D^i, B_1^i, \ldots, B_{k_i}^i \in P$ and $C \wedge D^i$ satisfiable

This is a different way of reducing the number of SLG-trees generated by the algorithm: SLG-trees are generated only for a subset of the predicate symbols in P. The remaining symbols are always resolved using program clauses, similarly to SLD-resolution. Again, soundness and completeness are preserved (by simple modification of Theorem 22). Termination is guaranteed if and only if at least one predicate is resolved by the SLG$^{\mathcal{C}}$ resolution for every cycle in the dependency graph of P (this follows by an easy extension of Theorem 23). Otherwise, an infinite branch may appear in some of the SLG-trees. This may lead to non-termination similarly to the case of SLD-resolution.

Also, as there is only a bounded number of SLD resolution steps between any two SLG$^{\mathcal{C}}$ resolution steps, the bodies of the non tabled clauses can be *unfolded* in the bodies of their callers. This transformation completely eliminates the need for non tabled resolution steps.

3. Program transformation similar to supplementary magic (Ramakrishnan, 1991). The previous folding transformation may introduce unnecessary recomputation of conjunctions of goals. This can be avoided by a technique that *folds* common parts of bodies of the clauses and creates separate clauses. Note that the recomputation is avoided by making the heads of such clauses tabled—resolved by the SLG$^{\mathcal{C}}$ resolution.

The last two optimizations are based on program transformations. However, in contrast to the Magic Transformation, these two transformations are completely query-independent.

4. Magic Templates Transformation for $\mathcal{C}$ (MT$^{\mathcal{C}}$)

This section describes a simple version of the program transformation approach to the goal-oriented query evaluation in constraint deductive databases—the Magic Templates transformation (in the constraint setting the difference between Magic Sets and Magic Templates is blurred). The transformation has to be slightly modified in the context of constraint databases.

Algorithm 24 (MT program transformation)

$$
\begin{aligned}
\mathrm{mst}(G, C, P) = \{\, & A \leftarrow D, \text{magic}A, B_1, \ldots, B_k, \\
& \text{magic}B_1 \leftarrow D, \text{magic}A, \\
& \quad\vdots \\
& \text{magic}B_k \leftarrow D, B_1, \ldots, B_{k-1}, \text{magic}A \;:\; A \leftarrow D, B_1, \ldots, B_k \in P \} \\
\cup \{\, & \text{magic}G \leftarrow C_i : C_i \in \exists_G C \}
\end{aligned}
$$

where magic$A$, magic$B_1, \ldots,$ magicB_k *are the magic atoms for* $A, B_1, \ldots, B_k$*, respectively.*

Again, for simplicity, only the left-to-right SIPS is used. This corresponds to the selection rule used in Constraint Memoing. In both cases, different selection rules may improve the efficiency of query evaluation (Ramakrishnan, 1991). However, in the case on MT, the SIPS is fixed during the *program transformation* phase and there are technical difficulties with combining different SIPS in one program. In the case of SLG$^{\mathcal{C}}$ evaluation, the selection rule can be adjusted during the evaluation process dynamically while preserving correctness of the answers.

The Magic Templates transformation is often preceded by an *adornment* transformation (Ramakrishnan, 1991). The adornment phase is designed to partition the search space according to (the statically derivable) information about free and bound arguments of the literals. The purpose of this transformation is threefold:

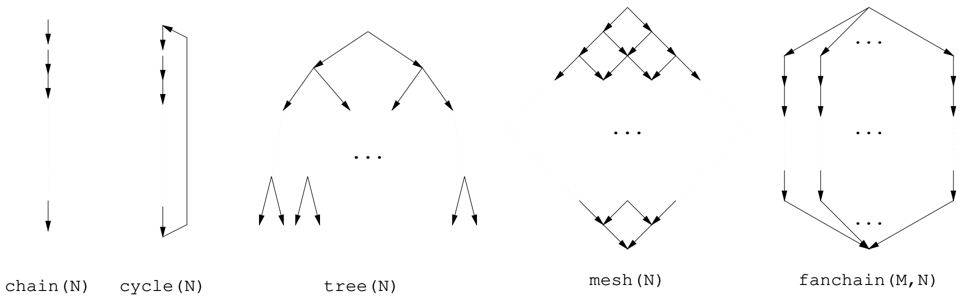
1. The original purpose of the adornments was to project out all the arguments that are not bound and thus guarantee that only ground tuples are generated (in the constraint setting this is not needed).
2. The other important consequence of using adornments is the possibility to reduce the arity of literals in the bodies of clauses. This leads to more efficient bottom-up evaluation: reduction of arity by one may cause linear speedup (Kemp et al., 1990).
3. The adornment partially factors the search space and allows to propagate only the needed restrictions.

In the SLG$^{\mathcal{C}}$ case the first use of adornments is not needed (similarly to the bottom-up procedure for constraints: Algorithm 9). Thus we implemented the MT without the adornment phase. The second and third uses are also partially achieved in the top-down evaluation: The factoring technique uses a static prediction of binding patterns of literals to reduce their arity. However, at the runtime, these literals have to follow this prediction and thus the effect of factoring is partially achieved using the *subsumption check*. Moreover, the run-time check detects *all* possible factoring opportunities (for the particular evaluation order) while the static methods can predict only a subclass of them. Also in many cases, the top-down method groups the answers to particular goals according the bindings present in these goals. This way it propagates only the relevant bindings (in the bottom-up method this effect could be achieved by building an goal-based index on the magic atoms). Note that this grouping of answers can not be achieved by using adornments as it depends on the actual data in the constraint database.

We test the relative performance of the evaluation methods by computing paths in various graphs.

```
tc(X,Z):-tc(X,Y),tc(Y,Z).     path(X,Z):-X<Y<Z,path(X,Y),path(Y,Z).
tc(X,Y):-e(X,Y).              path(X,Y):-e(X,Y).
```

In the gap-order case (`path`) we only look for paths where all edges lead from nodes with lower number to a node with higher number. We use the `tc` and `path` programs to find paths in the following graphs:



The nodes of the graphs are numbered by integers from top to bottom. In the case of gap-order Datalog, the integers are represented using constraints (e.g., i is represented by $i-1 < x < i+1$). Note that in the case of acyclic graphs the parent nodes are always labeled with a smaller integer.

Figure 3. Test Programs

5. Performance

In this section, the analytical complexity bound for the Constraint Memoing evaluation is discussed. The complexity of the top-down evaluation depends on the particular class of constraints $\mathcal{C}$; our analysis is based on the relative comparison to the (time- and space-) complexity of the bottom-up evaluation procedure. The analytical results are confirmed by experimental results that show the performance gains achieved by Constraint Memoing.

5.1. Theory

We show that the complexity of Constraint Memoing is no worse than complexity of the bottom-up evaluation:

Theorem 25 *Let* $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$ *be the result of the bottom-up evaluation of the query* (G, C, P) *and* $f(n)$ *a function such that* $|\,\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)| \in O(f(|P|))$. *Then*

1. *the SLG-forest* $\mathrm{SLG}(G, C)$ *has at most* $O(f^2)$ *nodes.*
2. *there are at most* $O(f^2)$ *applications of the SLG*$^{\mathcal{C}}$ *rules in the evaluation of the query.*

Proof: *(1) follows from the observation that every SLG-tree in the SLG forest has at most* $O(f)$ *leaves and thus also at most* $O(f)$ *nodes as the height of the trees is fixed by the* *number of subgoals in the clauses of P, and there are at most $O(f)$ different SLG trees in the SLG-forest as the size of $\mathrm{TP}_{\mathcal{C}}^{\omega}(\emptyset)$ limits the number of possible roots of the SLG-trees. (2) follows immediately from (1) as every application of a rule creates at least one new node in the SLG-forest.* ■

A careful implementation needs to store only a single path (of fixed length) in every SLG-tree in the SLG forest. Thus, the space requirements can be reduced to $O(f)$. On the other hand, the quadratic number of rule applications cannot be avoided in general. However, by analysis of the bottom-up algorithm the number of applications of *clauses* in P is also quadratic with respect to f (the bottom-up algorithm can recompute the same element of the interpretation several times, even if it is added only once to the interpretation $\mathrm{TP}_{\mathcal{C}}^{\omega}(\emptyset)$).

5.2. *Implementation*

We have implemented experimental versions of the following deductive query evaluation algorithms:

1. the Naive Bottom-up: a straightforward implementation of Algorithm 9,
2. the Semi-naive Bottom-up: a modification of Algorithm 9 (Ullman, 1989),
3. the Semi-naive Bottom-up with $\mathrm{MT}^{\mathcal{C}}$, and
4. the Constraint Memoing algorithm $\mathrm{SLG}^{\mathcal{C}}$.

The implementation of each of the evaluation algorithms is parametrized on the underlying class of constraints. For each constraint class we provide elementary operations on the constraint representation (cf. Definition 2) together with a few additional auxiliary operations:

`constraint_new(N,C)`: Given a number `N` it returns `C` as the representation of the constraint *true* over `N` variables. This operation is used to create a *fresh* environment for constraints, present during the evaluation of the individual clauses of the $\mathrm{Datalog}^{\mathcal{C}}$ program.

`constraint_and(G,C,CO,CN)`: This operation computes the constraint conjunction of the constraint `CO` with the constraint `C` where all the variables in `C` are renamed with respect to the variables of the atom `G`. This operation is used when a constraint derived by a subgoal of a clause is "and"-ed to the overall constraint over the variables in the body of the clause. The operation produces only consistent constraints; if the conjunction is not satisfiable, the operation fails.

`constraint_qe(G,C,Cn)`: Let `G` be an atom. Then `Cn` is a finite set (list) of constraints equivalent to the constraint `C` after all variables not in `G` are eliminated. This operation is used in the *Query Projection* rule, where we project the constraint on the free variables of the goal to be resolved, and the *Answer Projection* rule, where we eliminate all variables not present in the head of the clause.

`constraint_subsumes(C1,C2)` is the subsumption checking procedure. The operation succeeds if `C1` subsumes `C2`. We can assume that the constraints `C1` and `C2` are

Query	Data	Naive Bottom-up	Semi-naive Bottom-up	Semi-naive with MT	Top-down
tc(X,Y)	chain(32)	64590	40470	78790	29780
tc(1,32)	chain(32)	64250	40310	65910	2720
tc(1,24)	cycle(24)	108860	88040	102860	5740
tc(1,48)	tree(64)	48570	32120	70910	6730
tc(1,36)	mesh(6)	44370	19920	31940	1600
tc(0,37)	fanchain(2,18)	37560	17080	35400	1520
tc(0,37)	fanchain(6,6)	3530	2220	7330	610
tc(0,37)	fanchain(18,2)	1260	840	3200	490
path(X,Y)	chain(16)	16280	9790	31510	10780
path(1,16)	chain(16)	16300	9980	22020	5500
path(0,13)	fanchain(2,6)	16420	10300	37830	2700
path(0,17)	fanchain(4,4)	20390	10950	26230	2990
path(0,13)	fanchain(6,2)	5260	3480	14130	1790

Figure 4. Running times of test queries for various evaluation procedures (in msec).

over the same set of variables: we only use this operation to decide if a new constraint atom has been derived by the particular method or if a new SLG-tree is needed.

The last two operations are just for the convenience of the user of the system:

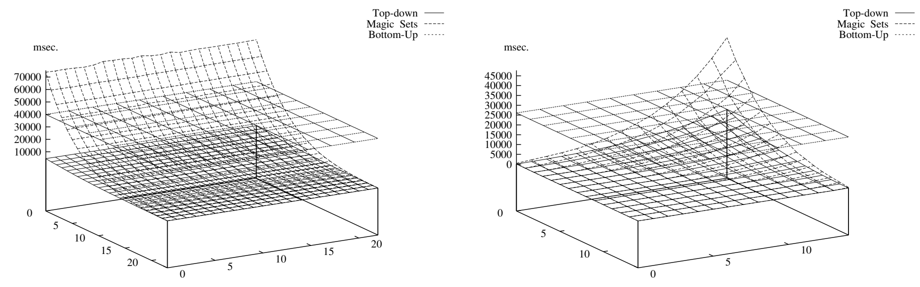
`constraint_pp(C)` allows "pretty printing" of the results of the evaluation, and

`constraint_read(T,G,C)` allows entering the constraints as formulas, rather than as the actual representation as Prolog terms.

In addition, we need to specify the Datalog$^{\mathcal{C}}$ programs that we intend to evaluate. Note that we use the *same* implementation of the operations on constraints for *all* the evaluation algorithms. Thus the relative performance of these algorithms is not caused by more sophisticated way of manipulating the constraint representation in one of the algorithms.

5.3. Experimental Results

Both the bottom-up (including the MT optimization) and the Constraint Memoing algorithms have been implemented in Prolog. We would like to emphasize that neither of the implemented evaluation algorithms takes an advantage of Prolog's top-down evaluation strategy—all the algorithms are implemented as meta-interpreters operating on a common ground representation of constraints (note that our top-down technique could have gained a considerable advantage by using Prolog's evaluation strategy). Figure 4 summarizes the running time of queries over graphs in Figure 3. The first line shows the performance for the uninstantiated case. In the constraint cases (`path`) the integers in the queries are expressed using constraints similarly to constants in Figure 3. The examples of the instantiated queries are those, where the optimization achieves the *least* effect (in all cases).



Datalog
over 24-element chain
`(tc, chain(24))`

Datalog with Gap-order constraints
over 16-element chain
`(path, chain(16))`

We measure the elapsed time to verify if there is a path from node i (x-axis) to node j (y-axis). The elapsed time is plotted on the z-axis. Note that in the ordered case (right graph), the constraint propagation allows more efficient pruning than in the case of standard Datalog (left graph). Thus the use of constraints may improve efficiency even for standard queries.

Figure 5. Elapsed time of query evaluation for all possible paths.

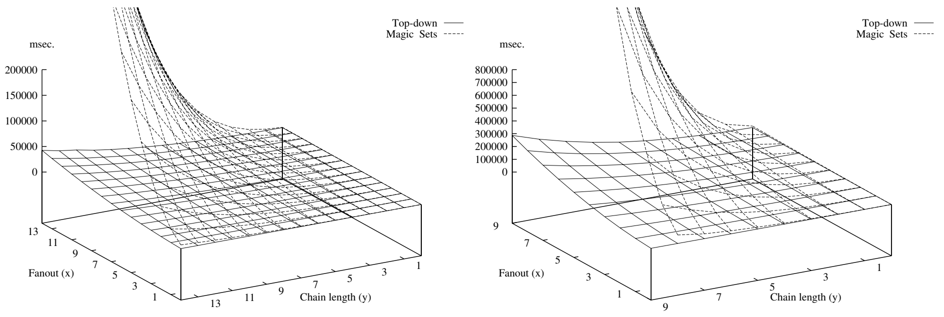
The results show that while the implementation of the various evaluation methods are comparable (the results on uninstantiated queries are approximately the same), the evaluation of instantiated queries is much more efficient using the Constraint Memoing algorithm. The boost is inherent to the top-down evaluation method is not caused by using a more sophisticated implementation. The other two experiments (Figures 5 and 6) show that the query evaluation on constraints generally follows the patterns of query evaluation on ground representation:

- for all possible queries over a given graph (Figure 5 plots the elapsed time for all ground queries $\mathtt{tc}(i, j)$ over a n-element chain. Similar graphs can be produced for the other structures in Figure 3), and
- for all *shapes* of the given graph (Figure 6 plots the elapsed time for graphs with varying fanout/fanin and chain lengths of $\mathtt{fanchain}(x, y)$ defined in Figure 3).

Thus we can expect very efficient Constraint Memoing-based query evaluation engines for constraint extensions of Datalog whose performance will be comparable to the top-down engines for ground Datalog (Swift and Warren, 1994a, Swift and Warren, 1994b).

6. Conclusion

We have proposed a practical approach to query evaluation for generalized constraint databases. Both the analytical and the empirical results show that Constraint Memoing is no worse than comparable bottom-up methods and in many cases the practical performance is much better even when using a very naive implementation. The performance



Datalog   Datalog$^{<Z}$

The above graphs plot the elapsed time to find a path in the `fanchain(x,y)` graph from the top-most node to the bottom-most node (cf. Figure 3). On the x and y axes we plot the parameters of the used graph: the fanout on the x-axis and the chain length on the y-axis. The elapsed time is plotted on the z-axis.

Figure 6. Elapsed time for varying fanout and chain length.

of the Constraint Memoing can be boosted by utilizing compilation methods developed in (Freire et al., 1996, Swift and Warren, 1994a, Swift and Warren, 1994b, Warren, 1983) and performance similar to ground Datalog can be expected. In addition, recent work on scheduling of operations in tabling systems (Freire et al., 1997) shows modifications to the tabling strategy that make it efficient even if external storage is involved. The scheduling strategies are orthogonal to the extensions introduced for handling constraints and thus can be immediately applied to our proposal.

6.1. Related Work

Recently, there have been several other attempts to make query evaluation in the presence of constraints efficient. There are two main directions of this research:

1. The first direction has its roots in the deductive database community: Techniques for pushing constraints present in the query were proposed in (Kemp and Stuckey, 1993, Ramakrishnan and Srivastava, 1993, Stuckey and Sudarshan, 1994). However, the goal of these methods is to *preprocess* the query (i.e., the goal and the rules) with respect to the given constraints for subsequent bottom-up evaluation. We present a completely different evaluation strategy where the constraints are propagated dynamically without the need for the preprocessing of the query. Also, in the standard database approach, the constraints are considered to be mere *conditions* that restrict the otherwise ground answers. Constraint Memoing uses constraints as a tool for *representing* both the data computed by the queries and stored in the database itself (i.e., non-ground relations are allowed). This dramatically increases the expressive power of the query language while preserving termination and efficiency.

2. The other direction is pursued in the area of (general) Logic Programming: In (Gao and Warren, 1993, Johnson, 1993, Lim and Stuckey, 1990) top-down evaluation for constraint logic programs is proposed. However, in all cases, general constraint solving procedures are used. Thus, these methods are not directly useful for query evaluation in constraint databases: termination of queries cannot be guaranteed. The closest to our work is (Gao and Warren, 1993). However, the method proposed there allows propagation of constants only (i.e., constraints of the form $x = a$); the *constraint* part of the query is essentially computed bottom-up. Our approach allows full propagation of all possible constraints during the whole evaluation process.

6.2. Directions of Future Research

Future research in this area needs to focus on the following issues:

1. Compilation of constraints. To achieve an efficient implementation of Constraint Memoing, data structures for efficient representation of the constraints have to be developed. There are two main differences to be addressed:

   - In most cases, the size of the constraint representation is bounded with respect to the arity of a literal. However, general Logic Programming engines allow unbounded terms to be built. Exploring this property may lead to an efficient *stack*-based implementation (i.e., without a heap) of the evaluation procedure.
   - On the other hand, classical Logic Programming assumes that every (logical) variable is either free or bound to a single term (and this binding can be changed only by backtracking). This assumption is no longer valid in the presence of constraints as more restrictive conditions may be derived after a variable is originally bound. Also, the constraints specify complex relations *between* the individual variables, which is not possible in the standard approach.

   Development of such a representation enables building of very efficient query evaluation engines based on partial evaluation of the atomic constraints in a given class, similarly to the WAM abstract code (Warren, 1983).
2. Analysis of binding patterns. Similarly to the MT transformation, the queries can be analyzed to determine the *flow of information* in clause bodies (Ramakrishnan, 1991). This is a considerably more complicated task in the presence of constraints: it is no longer sufficient to focus on single variables; the relationships between groups of variables have to be taken into consideration (as noted in Section 4). Also, the assumption that all EDB relations are ground (i.e., after resolution of an EDB goal all variables are bound to constants) is no longer valid—the generalized relations store representation of sets of tuples that may be infinite. Such an analysis can be used for several purposes: query optimization (MT-like rewriting), optimization of access to the constraint database (indexing), goal reordering, etc.
3. Interface to an existing RDBMS. As the constraints can be finitely encoded, their representation can be stored as tuples in a standard relational database system. However, query evaluation has to be carried out with respect to the semantics of such encoding (i.e., to perform, e.g., a join of two constraint relations, we can not use the join operation

of the underlying RDBMS directly). We propose the top-down evaluation procedure to be used as a front-end built on top of a standard relational DBMS. Similar approach was proposed in (Freire et al., 1996, Freire et al., 1997) for the XSB deductive system. The proposed techniques can be directly applied in the constraint setting.

4. Negation. Adding negation to Datalog$^{\mathcal{C}}$ in such a way that termination is preserved, is a nontrivial task: adding negation often leads immediately to Turing completeness. Essentially, adding negation (and preserving termination) would require the constraint class to be closed under negation (complementation) while preserving constraint-compactness. This condition is easy to satisfy in the case of finite domain constraints. For infinite constraint classes we sometimes need to restrict the class of allowed Datalog$^{\mathcal{C}}$ programs to those, where termination can be guaranteed (Revesz, 1995). (Toman, 1997) presents an extension of Constraint Memoing to Datalog$^{\mathcal{C}}$ programs with negation under the well founded semantics. However, the constraint class is required to be both constraint compact and closed under complementation.
5. Storage and access methods. To achieve an efficient implementation of constraint databases, new storage management techniques have to be developed: access methods suitable for fast retrieval of the stored information, efficient updates of generalized relations, indexing techniques (Kanellakis et al., 1993), etc.
6. Benchmarks. The performance of various implementations of Logic Programming languages (e.g., Prolog) is often judged by the performance on a standard benchmarks (e.g., `nrev`). We propose to develop similar benchmarks for query evaluation methods in constraint databases. The benchmarks should be independent of the particular class of constraints. Such test suite would allow us to compare performance of various query evaluation methods.

References

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. (1986). Magic Sets and Other Strange Ways to Implement Logic Programs. In *ACM Symposium on Principles of Database Systems*, pages 1–16.

Chen, W. and Warren, D. S. (1993). Query evaluation under the well-founded semantics. In *ACM Symposium on Principles of Database Systems*, pages 168–179.

Clocksin, W. F. and Mellish, C. S. (1987). *Programming in Prolog*. Springer, Berlin, 3 edition.

Freire, J., Swift, T., and Warren, D. S. (1996). Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 234–258.

Freire, J., Swift, T., and Warren, D. S. (1997). Taking i/o seriously: Resolution reconsidered for disk. In *International Conference on Logic Programming*.

Gao, H. and Warren, D. S. (1993). A powerful evaluation strategy for CLP programs. In *PPCP'93, First International Workshop on Principles and Practice of Constraint Programming*, pages 90–97.

Jaffar, J. and Maher, M. (1994). Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20):503–581.

Johnson, M. (1993). Memoization in constraint logic programming. In *PPCP'93, First International Workshop on Principles and Practice of Constraint Programming*, pages 130–138.

Kanellakis, P., Kuper, G., and Revesz, P. (1995). Constraint Query Languages. *Journal of Computer and System Sciences*, 51(1):26–52.

Kanellakis, P., Ramaswamy, S., Vengroff, D., and Vitter, J. (1993). Indexing for Data Models with Constraints and Classes. In *ACM Symposium on Principles of Database Systems*, pages 233–243.

Kemp, D., Ramamohanarao, K., and Somogyi, Z. (1990). Right-, left-, and multi-linear transformations that maintain context information. In *International Conference on Very Large Data Bases*, pages 380–391.

Kemp, D. B. and Stuckey, P. J. (1993). Analysis based constraint query optimization. In Warren, D. S., editor, *International Conference on Logic Programming*, pages 666–682.

Lim, P. and Stuckey, P. (1990). Meta programming as constraint programming. In *North American Conference on Logic Programming*, pages 416–430.

Lloyd, J. (1987). *Foundations of Logic Programming*. Springer-Verlag, 2nd edition.

Maher, M. (1993). A logic programming view of clp. In *International Conference on Logic Programming*, pages 737–753.

Ramakrishnan, R. (1991). Magic Templates: A Spellbinding Approach to Logic Programs. *Journal of Logic Programming*, 11(3&4):189–216.

Ramakrishnan, R. and Srivastava, D. (1993). Pushing Constraint Selections. *Journal of Logic Programming*, 16(3&4):361–414.

Revesz, P. (1993). A Closed-Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theoretical Computer Science*, 116:117–149.

Revesz, P. Z. (1995). Safe Stratified Datalog with Integer Order Programs. In *International Conference on Constraint Programming*, volume 1000 of *Lecture Notes in Computer Science*, pages 154–169.

Sagonas, K. F., Swift, T., and Warren, D. S. (1994). XSB as an efficient deductive database engine. In Snodgrass, R. T. and Winslett, M., editors, *ACM SIGMOD International Conference on Management of Data*, pages 442–453.

Srivastava, D. (1993). Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. *Annals of Mathematics and Artificial Intelligence*, 8:315–343.

Srivastava, D., Ramakrishnan, R., and Revesz, P. (1994). Constraint Objects. In Borning, A., editor, *PPCP'94, Second International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 181–192.

Stuckey, P. J. and Sudarshan, S. (1994). Compiling query constraints. In *ACM Symposium on Principles of Database Systems*, pages 56–67.

Swift, T. and Warren, D. S. (1994a). An abstract machine for SLG resolution: definite programs. In *Logic Programming - Proceedings of the 1994 International Symposium*, pages 633–652.

Swift, T. and Warren, D. S. (1994b). Analysis of SLG-WAM evaluation of definite programs. In *Logic Programming - Proceedings of the 1994 International Symposium*, pages 219–235.

Tamaki, S. and Sato, T. (1986). OLD Resolution with Tabulation. In *International Conference on Logic Programming*, pages 84–98.

Toman, D. (1995). Top-Down Beats Bottom-Up for Constraint Based Extensions of Datalog. In *International Logic Programming Symposium*, pages 189–203.

Toman, D. (1997). Computing the Well-founded Semantics for Constraint Extensions of Datalog$^{\neg}$. In *Constraint Databases and Applications*, volume 1191 of *Lecture Notes in Computer Science*, pages 64–79.

Toman, D., Chomicki, J., and Rogers, D. (1994). Datalog with Integer Periodicity Constraints. In *International Logic Programming Symposium*, pages 189–203.

Ullman, J. (1989). *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press.

Warren, D. H. D. (1983). An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA.

Williams, H. (1976). Fourier-Motzkin Elimination Extension to Integer Programming Problems. *Journal of Combinatorial Theory A*, 21:118–123.