A Framework for Bottom-Up Simulation of SLD-Resolution

STEFAN BRASS

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
(*e-mail:* brass@informatik.uni-halle.de)

submitted 14 February 2014; revised —; accepted —

Abstract

This paper introduces a framework for the bottom-up simulation of SLD-resolution based on partial evaluation. The main idea is to use database facts to represent a set of SLD goals. For deductive databases it is natural to assume that the rules defining derived predicates are known at "compile time", whereas the database predicates are known only later at runtime. The framework is inspired by the author's own SLDMagic method, and a variant of Earley deduction recently introduced by Heike Stephan and the author. However, it opens a much broader perspective.

KEYWORDS: deductive databases, Datalog, bottom-up evaluation, partial evaluation, optimization

1 Introduction

Deductive databases use logic programming for data intensive applications. For example, database queries are written in a Prolog-like language called Datalog. Basic Datalog is pure Prolog without structured terms. The data stored e.g. in a relational database can be seen as a large set of facts. Of course, many extensions of this basic language have been investigated and implemented in prototype systems. While in the beginning, the main achievement of deductive databases was seen in the possibility to write recursive queries, e.g. for hierarchical and graph-structured data, the more general goal is to support database queries and application programming in one declarative language.

For efficient query evaluation, it is important to distinguish between predicates defined by rules in a logic program, and predicates defined by facts in the database. The logic program is known at "compile time", while the facts are known only at "runtime". I.e. the database facts form the input to the logic program. Because the program might be executed several times with different database states, it pays off to invest time for optimization by precomputing as much as possible given the logic program, while the database facts are not yet known. Time might be saved even in a single execution of the program, because the logic program is usually small, while the database state is big.

Deductive databases use bottom-up evaluation, i.e. apply the T_P-operator to derive facts from already known facts to get logically implied instances of the query. Of course, many optimizations are added to this basic method. Especially, there are a lot of methods for making bottom-up evaluation goal-oriented, i.e. to derive only facts that are in some sense needed for computing answers to the query. Most well-known in this area is the "magic set" method (Bancilhon et al. 1986), where magic sets simply encode subqueries.

François Bry had the idea to explain magic sets with a meta-interpreter, which describes top-down evaluation, but runs on a bottom-up machine (Bry 1990). When this meta-interpreter is partially evaluated with respect to the given rules, one gets exactly the result of the magic set transformation. This might be the first case of partial evaluation for strict bottom-up evaluation, other applications of partial evaluation for deductive databases are, e.g. (Sakama and Itoh 1988; Lei et al. 1990; Han 1995). Of course, partial evaluation for top-down evaluation (Prolog) was well investigated (see, e.g., (Gallagher 1993)), but the methods for partial evaluation depend crucially on the execution model.

It turned out that magic sets do not exactly correspond to SLD-resolution, and that in the case of tail recursions, SLD-resolution has a big advantage, because it does not need to materialize every proven "lemma" (Ross 1991; Brass 1995).

The author then proposed a meta-interpreter that describes SLD-resolution exactly and runs on a bottom-up machine. In this way, set-oriented evaluation techniques can be used, and termination can be guaranteed for Datalog, e.g. a rule like $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{p}(\mathsf{X})$ does not cause an infinite loop. In contrast to magic sets and tabling techniques, tail-recursion runs much faster, e.g. consider the following logic program P:

$$\begin{array}{lcl} \mathsf{path}(\mathsf{X},\mathsf{Y}) & \leftarrow & \mathsf{edge}(\mathsf{X},\mathsf{Y})\cdot \\ \mathsf{path}(\mathsf{X},\mathsf{Z}) & \leftarrow & \mathsf{edge}(\mathsf{X},\mathsf{Y}) \land \mathsf{path}(\mathsf{Y},\mathsf{Z})\cdot \end{array}$$

Suppose the database is $\mathsf{D} := \{\mathsf{edge}(i-1, i) \mid 1 \le i \le n\}$, i.e. the graph is a single path of length n. For the query $\mathsf{path}(0, \mathsf{X})$, the SLD-tree has $4n+3$ nodes, i.e. a number that is linear in n, whereas magic sets derive all facts of the form $\mathsf{path}(i, j)$ with $0 \le i < j \le n$, which is quadratic in n. While a tail-recursion optimization for magic sets has already been studied in (Ross 1991), our "SLDMagic"-method (Brass 2000) had other advantages as well by simulating SLD-resolution bottom-up. It passes also non-equality conditions on parameters to called predicates, and avoids joins when a predicate "returns".

However, the possibilities for bottom-up simulation of SLD-resolution extend much farther if we allow a single fact to represent multiple nodes in the SLD-tree. Already when the SLDMagic method was implemented, it was noted that it produces a lot of "copy" rules, which only copy tuples from one predicate to another predicate. These rules were then eliminated by a postprocessing step, which merged the predicates. This can be understood as allowing facts to represent a set of goals in the SLD tree.

Recently, Heike Stephan and the author developed a variant of Earley deduction (Pereira and Warren 1983; Porter III 1986), also tabling (Tamaki and Sato 1986; Chen and Warren 1996; Nguyen and Cao 2012) can be seen as developing the Earley method further. Our Earley deduction variant uses states describing a relatively big part of deduction using only program rules, but no database facts (Brass and Stephan 2013). While there are differences in the technical details, this can be seen as representing a whole set of SLD goals in a single "state", which is encoded a a database fact.

The purpose of this paper is to introduce an abstract framework for simulating SLD-resolution on a bottom-up machine, improve the understanding of the mentioned methods and their similarities, and discuss options for improving the efficiency of program execution in deductive databases.

2 Basic Definitions

A logic program P is a finite set of rules of the form $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$ where A and B_i, $i = 1, \ldots, n$ (with $n \geq 0$) are positive literals, i.e. have the form $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_m)$ with a predicate p and terms t_j, $j = 1, \ldots, m$. Terms are variables or constants. We assume that the rules are range-restricted (safe), i.e. every variable appearing in the head A also appears in the body $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$. In this paper, we do not consider negation.

A subset of the predicates are selected as EDB predicates ("extensional database"). These are the database relations. The EDB predicates can appear in the logic program only in the body (i.e. in the B_i), but not in the head (A). Besides the logic program, a database state D is given, which is a finite set of facts (positive ground literals) in which only EDB predicates appear (it follows that P and D are disjoint). We write $\mathcal{B}$ for the set of all positive ground literals with EDB predicate (the Herbrand base restricted to EDB predicates). This set will usually be infinite (because it permits arbitrary strings, integers, etc. as arguments). Of course, $\mathsf{D} \subseteq \mathcal{B}$.

The predicates which do appear in rule heads are called IDB predicates ("intensional database"). These predicates are defined by means of rules, not by enumerating facts. It is possible that an IDB predicate has only rules with empty body (i.e. facts), then the only difference to an EDB predicate is that we assume that the program is given for partial evaluation ("compile time"), whereas the database is only known at runtime.

In practice, one also needs "built-in predicates" like $<$, which are defined by procedures inside the system. Such predicates raise interesting questions of range restriction and safety, see, e.g., (Ramakrishnan et al. 1987; Kifer et al. 1988; Brass 2009). However, to simplify the presentation, we exclude them here.

A query (goal) Q is a conjunction $\mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n$ of positive literals (like a rule body). The variables appearing in the query are called the answer variables. The purpose of query evaluation is to find ground substitutions for the answer variables such that the corresponding instance of the query is true in the minimal model of $\mathsf{P} \cup \mathsf{D}$.

To simplify the presentation, we assume the first literal selection rule for SLD-resolution (as applied in Prolog). In SLD-resolution, the computed answer substitution is normally defined by looking at an entire SLD-derivation leading to the empty clause:

$$Q \longrightarrow g_1 \longrightarrow g_2 \longrightarrow \cdots \longrightarrow g_n \longrightarrow \Box.$$

But it suffices to consider only single goals at a time, if we start with an "extended query", which has a literal with all answer variables and a special predicate answer at the very end: $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m)$. The predicate answer is not otherwise used in the program or the database. It only does the bookkeeping of the current values of the answer variables (since all substitutions done during the SLD derivation are also applied to this literal). Therefore, if we reach a goal consisting of a single literal $\mathsf{answer}(c_1, \ldots, c_m)$, we know that the answer substitution $\{X_1/c_1, \ldots, X_m/c_m\}$ has been computed.

It might seem at first that it is a restriction that we assume a completely given query. One might want to consider also "parameterized queries", containing constants not yet known at "compile time". For instance, one might want to do query optimization (partial evaluation) for any query of the form $\mathsf{path}(c, \mathsf{X})$, with an arbitrary constant c. This corresponds to "binding pattern" bf (mode $+-$) as used in the magic set method. However, we can simply use the query $\mathsf{input}(\mathsf{C}) \wedge \mathsf{path}(\mathsf{C}, \mathsf{X})$ with a database predicate input which is filled with the parameter value before query evaluation starts.

3 States Representing Sets of Goals

3.1 Definition of SLDDB-Systems

We first define an abstract system with states representing sets of goals in the SLD tree, and a transition relation between these states. The states will later be encoded as facts, and the transition relation corresponds to rules which permit to derive these facts.

A SLDDB-System $\mathcal{T}$ consists of

- $\mathcal{S}$, a (usually infinite) set of states,
- $\mathcal{G}(S)$, a non-empty set of goals for every $S \in \mathcal{S}$,
- $S_0 \in \mathcal{S}$, an initial state,
- $\mapsto_\epsilon \subseteq \mathcal{S} \times \mathcal{S}$, a relation between states (ϵ-transitions),
- $\mapsto_{\mathsf{F}} \subseteq \mathcal{S} \times \mathcal{S}$, a relation between states for every possible database fact $\mathsf{F} \in \mathcal{B}$.

Given an SLDDB-System $\mathcal{T}$ as above, and a database state $\mathsf{D} \subseteq \mathcal{B}$ (set of facts), an answer tuple $(c_1, \dots, c_m)$ is computed iff there is a finite sequence $S_0 \dots S_n \in \mathcal{S}^*$ of states, starting with the initial state S_0, containing the answer in the final state, i.e. $\mathsf{answer}(c_1, \dots, c_m) \in \mathcal{G}(S_n)$, and such that for $i = 1, \dots, n$,

- $S_{i-1} \mapsto_\epsilon S_i$ or
- $S_{i-1} \mapsto_{\mathsf{F}} S_i$ for some $\mathsf{F} \in \mathsf{D}$.

The states will later be encoded as facts, and the reachable states will be computed bottom-up, therefore repeated states in a state sequence can be detected in order to improve termination. Of course, if an analysis shows that this cannot happen, one can save the effort for duplicate detection. Another useful property is that for certain states, there is only one possible successor state (if D satisfies integrity constraints such as keys).

3.2 Correctness of SLD-Simulation

An SLDDB-System should correspond to SLD-resolution for a given program and query. First we define the correctness, i.e. that all goals occurring in a state sequence are really derivable from the query, the program, and the database facts used in state transitions.

Let a logic program P and a query Q be given. An SLDDB-System $\mathcal{T}$ is correct with respect to P and Q iff

- For every $g \in \mathcal{G}(S_0)$, there is an SLD-derivation of g from the extended query $Q \wedge \mathsf{answer}(\mathsf{X}_1, \dots, \mathsf{X}_m)$ using rules in P (this includes the case that the derivation is empty, i.e. g is the extended query).
- Whenever $S_1 \mapsto_\epsilon S_2$, then for every $g_2 \in \mathcal{G}(S_2)$ there is $g_1 \in \mathcal{G}(S_1)$ such that there is a non-empty SLD-derivation $g_1 \longrightarrow g' \longrightarrow^* g_2$ of g_2 from g_1, using rules in P, and the first step g' in this derivation is contained in $\mathcal{G}(S_2)$, i.e. $g' \in \mathcal{G}(S_2)$.
- Whenever $S_1 \mapsto_{\mathsf{F}} S_2$, then for every $g_2 \in \mathcal{G}(S_2)$ there is $g_1 \in \mathcal{G}(S_1)$ such that there is a non-empty SLD-derivation $g_1 \longrightarrow g' \longrightarrow^* g_2$ of g_2 from g_1, where the first step uses the fact F, other steps use rules in P, and $g' \in \mathcal{G}(S_2)$.

If this condition is satisfied, then for every computed answer tuple $(c_1, \dots, c_m)$ there is an SLD-derivation of $\mathsf{answer}(c_1, \dots, c_m)$ from $Q \wedge \mathsf{answer}(\mathsf{X}_1, \dots, \mathsf{X}_m)$. This obviously means that there is also an SLD-derivation of the empty clause from Q, where the substitution $\theta := \{\mathsf{X}_1/c_1, \dots, \mathsf{X}_m/c_m\}$ is applied to the variables in Q. Therefore, by the correctness of SLD-resolution, it follows that $Q\,\theta$ is a logical consequence of $\mathsf{P} \cup \mathsf{D}$.

3.3 Completeness of SLD-Simulation

In the opposite direction, we need that every SLD-derivation is indeed represented in the states and the transition relation. Note that the SLDDB-System is independent of a concrete database state, i.e. it represents derivations using any possible database facts. Of course, when a state sequence is constructed to answer a query in a concrete database state D, only facts in D can be used.

Let a logic program P and a query Q be given. An SLDDB-System $\mathcal{T}$ is complete with respect to P and Q iff

- The extended query $Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m)$ is contained in $\mathcal{G}(S_0)$.
- For every state $S \in \mathcal{S}$ and every goal $g \in \mathcal{G}(S)$ and every g' which can be derived from g and a rule in P by an SLD-resolution step, there is a variant g'' of g' with $g'' \in \mathcal{G}(S)$ or $g'' \in \mathcal{G}(S')$ for some state S' with $S \mapsto_\epsilon S'$.
- For every state $S \in \mathcal{S}$, every goal $g \in \mathcal{G}(S)$ and every g' which can be derived from g and a fact $\mathsf{F} \in \mathcal{B}$ by an SLD-resolution step, there is a variant g'' of g' and state S' with $S \mapsto_{\mathsf{F}} S'$ such that $g'' \in \mathcal{G}(S')$.

This condition ensures that every SLD-derivation of the empty clause from Q using rules in P and facts in DB can indeed be represented by a state sequence.

It would actually suffice to require the completeness not for all SLD resolution steps, but only for steps in successful derivations where the set of facts used in the state sequence satisfies given integrity constraints. In this way, “dead ends” could be cut off early.

3.4 Example: SLD-Resolution

Of course, one would expect that SLD-resolution itself fits in this framework.

If one wants to simulate SLD-resolution exactly, including the non-termination for $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{p}(\mathsf{X})$, one uses SLD-derivations as states. I.e., given a program P and a query Q, the set of states $\mathcal{S}$ is the set of SLD-derivations $Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m) \longrightarrow^* g_n$ and the set of goals for the above state S is $\mathcal{G}(S) := \{g_n\}$, i.e. a singleton set consisting of the last (or current) goal in the derivation. The transition relations extend the derivation by one goal, i.e. lead from S to the following state S':

$$Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m) \longrightarrow^* g_n \longrightarrow g_{n+1}.$$

If in the last SLD resolution step a program rule was used, $S \mapsto_\epsilon S'$. If instead a fact F with EDB-predicate was used, $S \mapsto_{\mathsf{F}} S'$.

3.5 Example: SLD-Resolution Without Duplicate Nodes

Of course, it is more in the spirit of bottom-up evaluation to eliminate duplicate nodes, and this is what SLDMagic (Brass 2000) did. In the above framework, this simply means that the states are single goals, i.e. conjunctions of the form

$$\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \mathsf{answer}(\mathsf{t}_1, \ldots, \mathsf{t}_m).$$

Furthermore, we have to exclude goals which differ only in a variable renaming. We do this by requiring that variables are named $\mathsf{V}_1, \mathsf{V}_2, \ldots$ in the order of first occurrence in the goal. Let $\mathsf{norm}(g)$ be a mapping from goals to goals which normalizes variables in this way. Of course, when states are single goals, we can simply let $\mathcal{G}(S) := \{S\}$. The transition relation is simply SLD-resolution plus the normalization. I.e. $S \mapsto_\epsilon S'$ holds iff S' is derivable from S by a single SLD resolution step (using a rule in P), followed by variable normalization. Correspondingly, $S \mapsto_\mathsf{F} S'$ holds when fact $\mathsf{F} \in \mathcal{B}$ is used in the resolution step. Obviously, the set of computed answers is not changed by merging nodes in the SLD-tree with the same goal. The number of duplicate answers is changed (every distinct answer is computed only once). However, if one considers duplicates as important, one probably wants a more declarative specification. Duplicates in deductive databases have been considered e.g. in (Mumick et al. 1990).

Note also that this works only because in SLD-resolution there is no need to return to the "caller" — otherwise the same subgoal could appear in different contexts, and it might be important to distinguish between them. This was one of the difficulties in our variant of Earley deduction. In SLD-resolution, the entire continuation of the proof is built into the goal. With the answer-literal at the end of the goal, even for determining the answer substitution, we do not have to look at an entire path in the tree.

Termination can be guaranteed if duplicate states in state sequences are excluded and the program is at most tail-recursive, i.e. only the last literal of a rule can be a recursive call. This property ensures that the length of goals is bounded (Brass 2000).

3.6 Maximal States

So far, states contained only single goals. But we want to compute as much as possible at "compile time", i.e. when query and program are known, but the facts in the database are not yet known. Therefore, we do the following closure operation on sets of goals:

$$\mathsf{cl}_\mathsf{P}(G) := \{\mathsf{norm}(g') \mid \text{there is } g \in G \text{ and an SLD-derivation } g \longrightarrow^* g' \text{ using only rules in } \mathsf{P}\}.$$

Now, given program P and query Q with answer variables $\mathsf{X}_1, \ldots, \mathsf{X}_m$, the initial state is the closure of the extended version of Q:

$$S_0 := \mathsf{cl}_\mathsf{P}\big(\{Q \wedge \mathsf{answer}(\mathsf{X}_1, \ldots, \mathsf{X}_m)\}\big).$$

Given a state S and a fact F with EDB-predicate, the successor state S' (with $S \mapsto_\mathsf{F} S'$) is defined as follows:

$$S' := \mathsf{cl}_\mathsf{P}\big(\{g' \mid \text{there is } g \in S \text{ such that a single SLD-resolution step of } g \text{ and } \mathsf{F} \text{ gives } g'\}\big).$$

The other transition relation $\mapsto_\epsilon$ is empty, i.e. state transitions occur only when EDB-facts are used, other deductions (with program rules) are done within the states.

Of course, it is possible that states become infinite. For instance, consider the left recursive version of the transitive closure example:

$$\begin{array}{lcl} \mathsf{path}(\mathsf{X},\mathsf{Y}) & \leftarrow & \mathsf{edge}(\mathsf{X},\mathsf{Y})\cdot \\ \mathsf{path}(\mathsf{X},\mathsf{Z}) & \leftarrow & \mathsf{path}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{edge}(\mathsf{Y},\mathsf{Z})\cdot \end{array}$$

Let the query be $\mathsf{path}(0,\mathsf{X})$. Then the initial state contains

$$\begin{array}{l} \mathsf{path}(0,\mathsf{V}_1) \wedge \mathsf{answer}(\mathsf{V}_1)\cdot \\ \mathsf{edge}(0,\mathsf{V}_1) \wedge \mathsf{answer}(\mathsf{V}_1)\cdot \\ \mathsf{path}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{answer}(\mathsf{V}_2)\cdot \\ \mathsf{edge}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{answer}(\mathsf{V}_2)\cdot \\ \mathsf{path}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{edge}(\mathsf{V}_2,\mathsf{V}_3) \wedge \mathsf{answer}(\mathsf{V}_3)\cdot \\ \mathsf{edge}(0,\mathsf{V}_1) \wedge \mathsf{edge}(\mathsf{V}_1,\mathsf{V}_2) \wedge \mathsf{edge}(\mathsf{V}_2,\mathsf{V}_3) \wedge \mathsf{answer}(\mathsf{V}_3)\cdot \\ \ldots \end{array}$$

However, this is not necessarily a problem if one can work with finite representations of this infinite set. For instance, the left recursive transitive closure works well in our variant of Earley deduction (Brass and Stephan 2013), and the graphs of partially processed rules used there can be seen as encoding an infinite set of SLD goals (if there are cycles in the "called by" relation). Furthermore, we have the following theorem, which shows that for programs without left recursions, the closure operation will not lead to infinite states:

Theorem 1
Suppose that P contains no IDB facts (i.e. all rules have a non-empty body), and no left recursions, i.e. the predicate of the first body literal of each rule does not depend on the predicate defined by the rule (i.e. it does not call — possibly indirectly — that predicate). Then $\mathsf{cl}_{\mathsf{P}}(G)$ is finite for every finite G.

Suppose for the moment that we can precompute all states (with parameters for the constants from database facts only known at runtime). Then working with maximal states is in some sense as efficent as it can get (when we look only at a single, successful derivation): One of the previously unknown database facts is processed in each step. If none of them is redundant (that depends on the program, e.g. there should be no repeated subgoals), any other query evaluation method must touch the same facts. But methods like "magic sets" also generate a lot of rules which do not contain EDB literals in the body. These rules are applied in addition to the necessary deductions with EDB literals. Furthermore, only a comparison with the magic set version with "supplementary predicates" would be fair (otherwise there are repeated accesses to the same fact in a derivation of a single answer). But then even more intermediate literals derived.

So, where is the hitch? At the moment, we can do the precomputation of states only for certain programs. Extending this set of programs is an interesting research problem.

Furthermore, there is a fundamental difference between SLD resolution and magic sets, namely, SLD resolution proves IDB literals always in the context of a concrete call (the goal contains everything that has to be done after the literal is proven), whereas magic sets derive ID predicates in isolation. Both has sometimes advantages: SLD resolution saves joins to get the proven literal back into the context of the caller, and with a more interesting selection function, conditions on the result can be checked at the best moment, and sometimes this is before the call is fully finished, see (Brass 2000). However, when magic sets have proven an IDB literal, they can use it several times in different contexts. SLD resolution (and thus our approach) has to prove it repeatedly. In the computation of a single answer this probably does not occur often, but when all answers are needed as in deductive databases, this might sometimes lead to suboptimal behaviour. In (Brass 2000) we proposed to mix both approaches, by doing explicit subproofs for certain literals. The same technique would work here. Ideally, we would have an automatic decision which of the two methods is better for a concrete call. This remains a research problem, too.

4 Encoding States as Facts

Our goal now is to study possibilities for encoding states as facts, such that the transition relation can be computed with standard Datalog rules. In this way, the approach becomes a source-to-source transformation like magic sets. However, the resulting rules have a very simple structure, such that other implementations, like a direct translation to C++, are an interesting option.

Such an encoding is also required because there is normally an infinite number of states: The SLDDB-system models deductions with all possible database facts, e.g. containing arbitrary strings as arguments. Often constants from the facts will be contained in the goals for continuing the proof, and therefore, the number of states is infinite. But for the precomputation at compile time, we need a finite representation of the set of states.

This is done by introducing parameterized states, which are mappings from a certain number of data values (the parameter values or arguments) to states. If we introduce a predicate for a parameterized state (with the arity equal to the number of parameters), we only have to define which concrete state is represented by a fact with this predicate.

Another way to see this is that at "compile time", we do not know the concrete constants from the database (only constants occurring in the program). Therefore, we represent these unknown values by "parameters". Later, at "runtime", we have values for the parameters, and can fill in the "holes" to get a fully specified state.

However, simply replacing parameters in the goals by constants is not the only way how states can be encoded. As explained below, the counting method can be understood as encoding the length of a part of a goal (of very regular structure) in a parameter value. This permits to handle (at least some) cases where the length of the occurring goals depends on the data, therefore, we cannot precompute them at compile time. Note that this is different from the left recursive version of the transitive closure discussed above: There, we had goals of arbitrary length in a single state, and therefore did not need to store any concrete length. In the programs, for which the counting method is made (especially the same generation example), the exact length is important.

4.1 Parameters for Constants Known Only at Runtime

Let us first consider the case where the parameters are simply replaced by constants. We assume that there are special variables $\mathsf{C}_1, \mathsf{C}_2, \ldots$ for the parameters (disjoint from the variables $\mathsf{V}_1, \mathsf{V}_2, \ldots$ used for normalization). Now a parameterized state with n parameters is defined by a set of goals in which the variables $\mathsf{C}_1, \ldots, \mathsf{C}_n$ can occur, and the variables $\mathsf{V}_1, \mathsf{V}_2, \ldots$, but no other variables. Furthermore, each goal must satisfy the normalization requirement, i.e. if the variable V_i, $i > 1$, occurs somewhere in a goal, all variables $\mathsf{V}_1, \ldots, \mathsf{V}_{i-1}$ must occur to the left in the same goal. Whereas the scope of standard variables $\mathsf{V}_1, \mathsf{V}_2, \ldots$ is only a single goal, i.e. they are a kind of "local variables", parameters are "global" in the parameterized state (set of goals). Therefore, a normalization is more difficult (we must use a standard order of goals), but of course, we do not construct distinct states which differ only in a renaming of the parameters. Now, if a parameterized state G with n parameters is encoded as predicate p, then the fact $\mathsf{p}(c_1, \ldots, c_n)$ represents $\big\{ g\,\theta \bigm| g \in G,\ \theta = \{\mathsf{C}_1/c_1, \ldots, \mathsf{C}_n/c_n\} \big\}$.

The unification procedure must be changed in order to respect the parameters. We must keep in mind that at runtime, there will be constants for them. The first change is that if we need to unify a parameter and a standard variable, we replace the variable by the parameter. The second change is that when we need to unify two parameters, or a parameter and a constant, unification produces a condition for the parameter values, e.g. $\mathsf{C}_i = a$. Thus, the unification procedure does not only yield a substitution for the standard variables, but also a (consistent) conjunction of conditions for the parameters. For each call to the unification procedure, we must make a case distinction. Either the actual parameter values satisfy the condition (e.g. $\mathsf{C}_i = a$), and the unification succeeds, or they do not ($\mathsf{C}_i \neq a$), and the unification fails. Note that when we work with sets of goals, some failed unifications do not necessarily lead to the empty result set. If the computation of the successor state needs k unifications, there could be 2^k cases to distinguish, but usually it will be much less, because we can stop as soon as the condition which describes the case becomes inconsistent. Each case might yield a different parameterized state. In the rule that describes the state transition, we have to check the conditions on the parameters, and also the non-equality conditions in order to avoid computing the same SLD derivation twice. See also (Brass and Stephan 2013).

4.2 Other Encodings: Counting

For general recursions, goals might become larger and larger depending on the data, thus it is not possible to precompute them explicitly at compile time (even if we replace unknown constants by parameters). E.g., this happens in the “same generation” example:

$$\begin{array}{lll} \mathsf{sg}(\mathsf{X},\mathsf{X}) & \leftarrow & \mathsf{person}(\mathsf{X})\cdot \\ \mathsf{sg}(\mathsf{X},\mathsf{Y}) & \leftarrow & \mathsf{parent}(\mathsf{X},\mathsf{X}') \wedge \mathsf{sg}(\mathsf{X}',\mathsf{Y}') \wedge \mathsf{parent}(\mathsf{Y},\mathsf{Y}')\cdot \end{array}$$

However, other types of encodings are possible. For instance, when applying the counting method (Bancilhon et al. 1986; Greco and Zaniolo 1992) to the same generation example, one can view $\mathsf{c\_sg}(\mathsf{C},\mathsf{I})$ as representing

$$\mathsf{sg}(\mathsf{C},\mathsf{Y}_1) \wedge \mathsf{parent}(\mathsf{Y}_2,\mathsf{Y}_1) \wedge \cdots \wedge \mathsf{parent}(\mathsf{Y}_{\mathsf{I}+1},\mathsf{Y}_\mathsf{I}) \wedge \mathsf{answer}(\mathsf{Y}_{\mathsf{I}+1})\cdot$$

5 Conclusions

This paper offers a different view on some previous methods for query evaluation, such as SLDmagic, counting, and a variant of Earley deduction. By introducing a common framework for them, one can compare and combine their features, and this also opens a space for thinking about new, improved methods.

Obviously, there are currently still many (interesting) research questions, and few readymade methods beyond what was already there. Nevertheless, the understanding ist improved, and the potential of the presented ideas seems promising.

Currently, a prototype implementation for the method sketched in Sections 3.6 and 4.1 is being developed (for the class of programs which are at most tail recursive). See

`http://www.informatik.uni-halle.de/~brass/slddb/`.

As a further generalization of the framework, one could start subproofs for certain literals and and possibly reuse their results multiple times, in order to get magic sets, see (Brass 2000). It might also be possible to split goals in pieces and link them together by references to previous states, somewhat similar to (Greco and Zaniolo 1992).

Acknowledgements

I would like to thank Heike Stephan for starting the research on Earley deduction, which inspired the abstraction presented here. I would also like to thank Marcus Lehmann for doing performance tests with the SLDmagic method.

References

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. 1986. Magic sets and other strange ways to implement logic programs. In *Proc. of the 5th ACM Symp. on Principles of Database Systems (PODS'86)*. ACM Press, 1–15.

Brass, S. 1995. Magic sets vs. SLD-resolution. In *Advances in Databases and Information Systems (ADBIS'95)*, J. Eder and L. A. Kalinichenko, Eds. Springer, 185–203.

Brass, S. 2000. SLDMagic — the real magic (with applications to web queries). In *First International Conference on Computational Logic (CL'2000/DOOD'2000)*, W. Lloyd et al., Eds. Number 1861 in LNCS. Springer, Heidelberg, Berlin, 1063–1077.

Brass, S. 2009. Range restriction for general formulas. In *23rd Workshop on (Constraint) Logic Programming (WLP'09)*, A. Wolf and U. Geske, Eds. Universitätsverlag Potsdam, 125–137.

Brass, S. and Stephan, H. 2013. A variant of earley deduction with partial evaluation. In *Datalog ieasoning and Rule Systems - 7th International Conference, RR 2013*, W. Faber and D. Lembo, Eds. LNCS, vol. 7994. Springer-Verlag, 35–49.

Bry, F. 1990. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering 5*, 289–312.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1, 20–74.

Gallagher, J. P. 1993. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'93)*. ACM, 88–98.

Greco, S. and Zaniolo, C. 1992. Optimization of linear logic programs using counting methods. In *Advances in Database Technology — EDBT'92, 3rd Int. Conf.*, A. Pirotte, C. Delobel, and G. Gottlob, Eds. Number 580 in LNCS. Springer-Verlag, 72–87.

Han, J. 1995. Chain-split evaluation in deductive databases. *IEEE Transactions on Knowledge and Data Engineering 7,* 2, 261–273.

Kifer, M., Ramakrishnan, R., and Siberschatz, A. 1988. An axiomatic approach to deciding query safety in deductive databases. In *Proc. of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'88)*. 52–60.

Lei, L., Moll, G.-H., and Kouloumdjian, J. 1990. A deductive database architecture based on partial evaluation. *SIGMOD Record 19,* 3, 24–29.

Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In *Proc. of the 16th International Conf. on Very Large Data Bases (VLDB'90)*, D. McLeod, R. Sacks-Davis, and H.-J. Schek, Eds. Morgan Kaufmann, 264–277.

Nguyen, L. A. and Cao, S. T. 2012. Query-subquery nets. In *Computational Collective Intelligence. Technologies and Applications. 4th International Conference, ICCCI 2012, Proceedings, Part I*, N.-T. Nguyen et al., Eds. Number 7653 in LNCS. Springer, 239–248.

Pereira, F. C. N. and Warren, D. H. D. 1983. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics (ACL)*. 137–144.

Porter III, H. H. 1986. Earley deduction. `http://web.cecs.pdx.edu/~harry/earley/`.

Ramakrishnan, R., Bancilhon, F., and Siberschatz, A. 1987. Safety of recursive horn clauses with infinite relations. In *Proc. of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87)*. 328–339.

Ross, K. A. 1991. Modular acyclicity and tail recursion in logic programs. In *Proc. of the Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Princ. of Database Systems (PODS'91)*. 92–101.

Sakama, C. and Itoh, H. 1988. Partial evaluation of queries in deductive databases. *New Generation Computing 6*, 249–258.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proc. Third Int. Conf. on Logic Programming (ICLP)*, E. Shapiro, Ed. Number 225 in LNCS. Springer, 84–98.