A Variant of Earley Deduction with Partial Evaluation*

Stefan Brass and Heike Stephan

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

Abstract. We present an algorithm for query evaluation given a logic program consisting of function-free Datalog rules. The algorithm is based on Earley Deduction [7, 10], but uses explicit states to eliminate rules which are no longer needed, and partial evaluation to minimize the work at runtime. At least in certain cases, the new method is more efficient than our SLDMagic-method [2], and also beats the standard Magic set method. It is also theoretically interesting, because it consumes one EDB fact in each step. Because of its origin, it is especially well suited for parsing applications, e.g. for extracting data from web pages. However, it has the potential to speed up basic Datalog reasoning for many semantic web applications.

1 Introduction

The goal of deductive database systems is to support efficient reasoning with large numbers of facts. This is for instance needed in semantic web applications. Although there is a relatively long history of research for query evaluation in deductive databases, with the magic set method as an established standard, efficiency is still a problem. This is not only a matter of finding a competent programmer, but it also needs more research. For instance, in [2] we proposed an improvement to the magic set method, and in [3] we studied implementation alternatives for bottom-up evaluation.

In the last time, extensions of Datalog have been shown to be very relevant for semantic web tasks, e.g. [8] (see also [1, 9]) shows how to translate SPARQL to Datalog with answer set semantics, [6] translates description logics into variants of Datalog with existential quantification in the rule heads, and [5] discusses how to adapt Datalog to semistructured data and to RDF. In [4], we study how output, e.g. the generation of web pages, can be done declaratively with an extension of Datalog with ordered predicates.

All these extensions of course need also the basic reasoning capability for classical Datalog, which is the subject of this paper. We take an old algorithm for query evaluation, Earley Deduction [7, 10], and improve it by separating

* This is a significantly extended and improved version of a paper that appeared in the 26th Workshop on Logic Programming (WLP'12).

states, eliminating unnecessary rules early, and doing partial evaluation. The partial evaluation is somewhat similar to the one we developed for our SLDMagic method [2]. It makes the algorithm very competitive, because a lot can already be done at compilation time.

The new algorithm loops through a sequence of states (basically sets of rules being processed), where a single database fact is used from one state to the next.

The algorithm is interesting also because it especially fits applications in which input must be parsed (after all, the Earley algorithm was a parsing algorithm). While our modified Earley deduction can in principle be used for arbitrary logic programs, there is a special optimization potential when it can be proven that only one fact is applicable in a state, and we do not have to check whether there is a cycle in the state sequence. In parsing applications, the EDB facts have a natural order, and are consumed in this order.

2 Basic Definitions

Definition 1 (Rule). *A rule is a formula of the form* $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$*, where* A *and* B_i*,* $i = 1, \ldots, n$ *are positive literals, i.e. atomic formulas* $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_m)$ *with a predicate* p *and terms* t_j*,* $j = 1, \ldots, m$*. Terms are variables or constants. The head of the above rule is* A*, the body is* $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$*. A rule with empty body (i.e.* $n = 0$*) and without variables is called a fact.*

Definition 2 (Range Restriction). *A rule is range-restricted iff every variable that appears in the head appears also in the body.*

Range restriction ensures that when a rule is applied to derive an instance of the head (given facts matching the body literals), the derived instance does not contain variables, i.e. is again a fact.

Definition 3 (EDB- and IDB-Predicates, Program and Database). *Predicates are partioned into EDB ("extensional database") and IDB-predicates ("intentional database"). A logic program* P *is a finite set of range-restricted rules with an IDB-predicate in the head and a non-empty body. A database* D *is a finite set of facts with EDB-predicate.*

The requirement that the body of program rules is non-empty simplifies later definitions, but is no restriction: One can use an EDB-predicate true without arguments and put it into the database.

Definition 4 (Answer Predicate). *We assume that an IDB-predicate* answer *is distinguished as "main" predicate. It may appear only in the heads of rules in the program.*

We assume that the program contains the query in the form of rules about this special predicate. The goal of query evaluation is to determine the answer-facts which are derivable from $\mathsf{P} \cup \mathsf{D}$, i.e. program and database together. Besides rules about the answer predicate, the program may also contain view definitions or rules defining auxiliary predicates.

Definition 5 (Selection Function). *A selection function chooses for every rule* $\mathsf{A} \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_n$ *with* $n \geq 1$ *an index* $i \in \{1, \ldots, n\}$ *(i.e. a body literal).*

For simplicity of presentation, in the following we will assume a selection function that always selects the first body literal. However, we note that in the database context the selection function is an important optimization parameter. So a real implementation will use a selection function that tries to make use of input arguments (constants) and database access structures (indexes).

Definition 6 (Normalization of Rules). *Let an infinite sequence* $\mathsf{X}_1, \mathsf{X}_2, \ldots$ *of variables be given. Without loss of generality we assume that these variables do not appear in the given program. A rule* R *is called normalized iff it contains only variables that appear in this sequence, and if* X_i *appears in the rule, then* $\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}$ *must appear to the left of* X_i *in the rule (i.e. variables are numbered in the order of first occurrence).*

The normalized version of a rule R' *is the unique normalized rule* $R = R'\,\theta$ *which can be reached by a variable renaming* θ*.*

The condition that the X_i do not appear in the program saves us a variable renaming when unifying literals from a normalized rule and a program rule.

3 Deduction Method

The method works by looping through a series of states, where one EDB fact is processed in each state transition. Each state contains the rules or remainders of rules which are currently being processed, i.e. which can participate in the derivation of answer-facts after having seen a certain sequence of database facts. Body literals which are already proven are removed, and unifying substitutions are applied to the rules. Therefore the rules in the state are specialized and simplified versions of program rules. Within the state, we explicitly represent the "is called by" relation between the rules:

Definition 7 (State). *A state* $\mathcal{S}$ *is a finite directed graph* $(\mathcal{V}, \mathcal{E})$*, the nodes* $\mathcal{V}$ *of which are normalized rules, and an edge* $(R_1, R_2) \in \mathcal{E}$ *is read "*R_1 *has been called by (the selected literal in)* R_2*".*

During the construction of a state, we need in addition the information which rules are new, because certain deduction steps are only applied to new rules:

Definition 8 (Extended State). *An extended state is a triple* $(\mathcal{V}, \mathcal{E}, \mathcal{A})$ *such that* $(\mathcal{V}, \mathcal{E})$ *is a state, and* $\mathcal{A} \subseteq \mathcal{V}$*. Rules in the set* $\mathcal{A}$ *are called "active rules".*

The construction of the initial state starts with (normalized versions of) all rules about answer, which are all active. There are no edges yet, because these rules did not call each other (answer appears only in the rule heads).

The following operation adds called program rules. Input values from the call are propagated to the rule by adding not the rule itself, but a specialized version of the rule, such that all derived instances of the head will fit the calling literal:

Definition 9 (Instantiation). *Let an extended state $\mathcal{S} = (\mathcal{V}, \mathcal{E}, \mathcal{A})$ and a program P be given. For each active rule $R_1 \in \mathcal{A}$ and program rule $R_2 \in \mathsf{P}$ such that the head of R_2 unifies with the selected literal in R_1, let θ be the most general unifier, and R_2' be the normalized version of $R_2\,\theta$. Then the extended state $\mathcal{S}' = (\mathcal{V}', \mathcal{E}', \mathcal{A}')$ with $\mathcal{V}' := \mathcal{V} \cup \{R_2'\}$, $\mathcal{A}' := \mathcal{A} \cup \{R_2'\}$, and $\mathcal{E}' := \mathcal{E} \cup \{(R_2', R_1)\}$ can be reached by instantiation. If $\mathcal{S}' \neq \mathcal{S}$, we write $\mathcal{S} \to_I \mathcal{S}'$.*

The following operation does a resolution step with a given EDB fact: It removes the corresponding body literal and applies the unifying substitution to the rest of the rule. This operation is called "reduction".

Definition 10 (EDB-Reduction). *Let an extended state $\mathcal{S} = (\mathcal{V}, \mathcal{E}, \mathcal{A})$ be given. For each rule $R \in \mathcal{V} - \mathcal{A}$, $R = \mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$, such that F unifies with the selected literal B_1, let θ be the mgu, and R' be the normalized version of $(\mathsf{A} \leftarrow \mathsf{B}_2 \wedge \cdots \wedge \mathsf{B}_n)\,\theta$. Then the extended state $\mathcal{S}' = (\mathcal{V}', \mathcal{E}', \mathcal{A}')$ with $\mathcal{V}' := \mathcal{V} \cup \{R'\}$, $\mathcal{A}' := \mathcal{A} \cup \{R'\}$, and $\mathcal{E}' := \mathcal{E} \cup \{(R', R'') \mid (R, R'') \in \mathcal{E}\}$ can be reached by EDB-reduction with F. If $\mathcal{S}' \neq \mathcal{S}$, we write $\mathcal{S} \to_{E(\mathsf{F})} \mathcal{S}'$.*

The condition that EDB-reduction can be applied only to non-active rules avoids that the same rule can be reduced twice in one state transition. This is important to keep the facts used in state transitions synchronized with the EDB fact sequence (see Def. 17).

When reduction has removed the last body literal, an IDB fact is proven. Then the called rule returns the answer to the caller, and the IDB fact is used to reduce the body of the calling rule. The difference to EDB-reduction is that here the "is called by" relation between the rules is used (EDB reduction reduces any rule in the state, IDB reduction only the calling rule).

Definition 11 (IDB-Reduction). *Let an extended state $\mathcal{S} = (\mathcal{V}, \mathcal{E}, \mathcal{A})$ be given. For each pair of rules $(R_1, R_2) \in \mathcal{E}$, such that R_1 is a fact F, and R_2 has the form $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$, where the selected literal B_1 is unifiable with F, let θ be the mgu, and R' be the normalized version of $(\mathsf{A} \leftarrow \mathsf{B}_2 \wedge \cdots \wedge \mathsf{B}_n)\,\theta$. Then the extended state $\mathcal{S}' = (\mathcal{V}', \mathcal{E}', \mathcal{A}')$ with $\mathcal{V}' := \mathcal{V} \cup \{R'\}$, $\mathcal{A}' := \mathcal{A} \cup \{R'\}$, and $\mathcal{E}' := \mathcal{E} \cup \{(R', R_3) \mid (R_2, R_3) \in \mathcal{E}\}$ can be reached by IDB-reduction. If $\mathcal{S}' \neq \mathcal{S}$, we write $\mathcal{S} \to_R \mathcal{S}'$.*

Actually, the states are constructed such that when there is an edge from R_1 to R_2, the head of R_1 is always unifiable with the selected literal in R_2: When a rule is added by instantiation, its head can be only more specialized than the calling literal (one would get it by applying the unifier to the selected literal in R_2). Later reduction steps specialize R_1 further, while R_2 does not change.

So far, we have defined single deduction steps. But of course, we want to apply them repeatedly "until nothing changes", so we reach a kind of fixpoint. For a given fact $\mathsf{F} \in \mathsf{D}$, let the combined relation be $\to_{C(\mathsf{F})} := \to_I \cup \to_{E(\mathsf{F})} \cup \to_R$.

Note that all of the above relations between extended states are confluent, i.e. if $\mathcal{S} \to^* \mathcal{S}_1$ and $\mathcal{S} \to^* \mathcal{S}_2$ there is a state $\mathcal{S}'$ with $\mathcal{S}_1 \to^* \mathcal{S}'$ and $\mathcal{S}_2 \to^* \mathcal{S}'$. This is obvious because the operations only add nodes and edges to the state,

and additional nodes and edges do not prevent the application of an operation that was applicable before.

For a given program, database, and state, instantiation and reduction can be applied only finitely many times, because only instances of program rules or "remainders" of rules already in the state can be added: No rule can get longer than the longest rule in the state and the program, and no rule can contain other constants than the constants contained in state, program and database. The variables in the rules are normalized, so the number of variables appearing in the state is also bounded by the number of terms in the longest rule.

Therefore, every state has a unique normal form with respect to each of these relations. The normal form can be reached by iteratively applying the relation as long as possible (i.e. until nothing more can be added to the state). We need two normal forms, one with respect to instantiation only, and one applying all deduction steps (given a single database fact):

Definition 12 (Closures). *Given an extended state $\mathcal{S}$ and program P, we write $\mathsf{Inst}^+(\mathcal{S})$ for the unique normal form of $\mathcal{S}$ with respect to $\to_I$ (instantiation).*

Given an extended state $\mathcal{S}$, program P, and a fact $\mathsf{F} \in \mathsf{D}$, we write $\mathsf{Conseq}^+_{\mathsf{F}}(\mathcal{S})$ for the unique normal form of $\mathcal{S}$ with respect to $\to_{C(\mathsf{F})}$.

Definition 13 (Initial State). *Given a program P, let $\mathcal{V}$ be the set of answer rules: $\mathcal{V} := \{R \in \mathsf{P} \mid R \text{ is a rule about } \mathsf{answer}\}$. Let $\mathcal{S}_0 := \mathsf{Inst}^+((\mathcal{V}, \emptyset, \mathcal{V}))$ (i.e. instantiation is applied to the given initial rules, which are all considered active). If $\mathcal{S}_0 = (\mathcal{V}_0, \mathcal{E}_0, \mathcal{A}_0)$, the initial state for this program is $(\mathcal{V}_0, \mathcal{E}_0)$.*

I.e. the initial state consists of the rules about answer in P, plus all rules which can be added by instantiation (repeatedly, until nothing changes).

Now the successor state of a state $(\mathcal{V}, \mathcal{E})$ is computed by applying all deduction steps with a given database fact F, i.e. computing $\mathsf{Conseq}^+_{\mathsf{F}}(\mathcal{S})$ with $\mathcal{S} := (\mathcal{V}, \mathcal{E}, \emptyset)$. (All rules are considered inactive at the beginning, but active rules are generated via EDB-reduction.) After the consequences are computed, we need the following "cleanup" operation, which removes rules which are no longer needed. Remember that reduction only adds new rules, but it does not remove the original rules. This is done for two reasons: (1) It ensures the confluence, and thus the existence of a unique normal form. (2) In the case of IDB reduction, the rule might permit several distinct reductions, either in the same state transition, or in a later state transition. So we must be careful to remove only rules which cannot contribute to a solution in this state sequence.

Definition 14 (State Cleanup). *Given an extended state $\mathcal{S} = (\mathcal{V}, \mathcal{E}, \mathcal{A})$, let $\mathcal{V}'$ be the set of rules $R \in \mathcal{V}$ satisfying one of the following conditions:*

- *R is a fact about answer, or*
- *R is reachable via the edges in $\mathcal{E}$ from an active rule $R' \in \mathcal{A}$ (this includes the case $R \in \mathcal{A}$), where R' has a non-empty body (this implies that R has a non-empty body, too).*

Then $\mathsf{Cleanup}(\mathcal{S}) := (\mathcal{V}', \mathcal{E}')$ with $\mathcal{E}' := \mathcal{E} \cap (\mathcal{V}' \times \mathcal{V}')$.

IDB-facts are removed, because all possible derivations with them have been done. The answer-facts are treated specially, because the "output" computed in the state needs to be stored somewhere. However, they will not participate in further derivations, so except for the output, they are not needed in the state.

Definition 15 (Successor State). *Let a program* P*, a database* D*, a state* $\mathcal{S} = (\mathcal{V}, \mathcal{E})$*, and a fact* $\mathsf{F} \in \mathsf{D}$ *be given. The successor state* $\mathcal{S}'$ *for* $\mathcal{S}$ *with respect to* F *is* $\mathsf{Cleanup}(\mathsf{Conseq}^+_{\mathsf{F}}((\mathcal{V}, \mathcal{E}, \emptyset)))$*, unless the result is empty, in which case there is no successor state.*

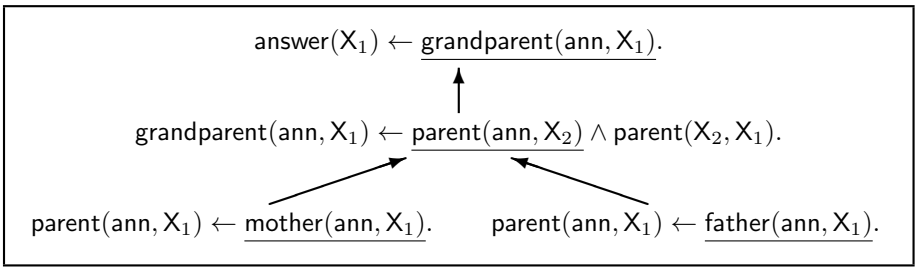
Definition 16 (Computed Answers). *A fact* $\mathsf{answer}(c_1, \ldots, c_m)$ *is computed if there is a sequence of states* $\mathcal{S}_0, \ldots, \mathcal{S}_n$ *such that* $\mathcal{S}_0$ *is the initial state,* $\mathcal{S}_i$ *is the successor state for* $\mathcal{S}_{i-1}$ *with respect to a fact* $\mathsf{F}_i \in \mathsf{D}$ *(*$i = 1, \ldots, n$*), and* $\mathsf{answer}(c_1, \ldots, c_m) \in \mathcal{V}_n$*, where* $(\mathcal{V}_n, \mathcal{E}_n) = \mathcal{S}_n$*.*

As explained above, there can be only finitely many states for a given program P and database D. However, the sequence of states could be cyclic, so one must check whether a newly constructed state is indeed new. Of course, optimizations are possible and subject of our further research.

Example 1. Let the following program be given:

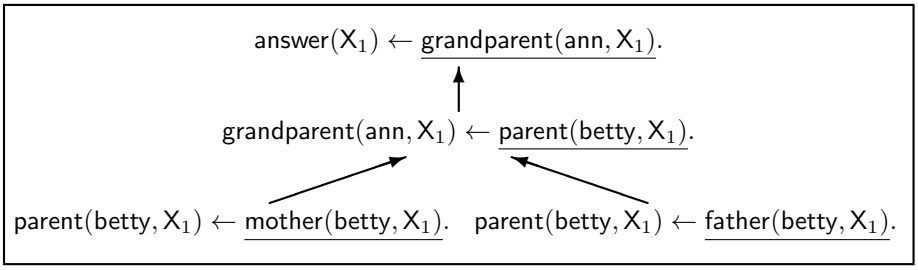
$$\begin{array}{ll}
\mathsf{grandparent}(\mathsf{X}, \mathsf{Z}) & \leftarrow \mathsf{parent}(\mathsf{X}, \mathsf{Y}) \wedge \mathsf{parent}(\mathsf{Y}, \mathsf{Z}). \\
\mathsf{parent}(\mathsf{X}, \mathsf{Y}) & \leftarrow \mathsf{mother}(\mathsf{X}, \mathsf{Y}). \\
\mathsf{parent}(\mathsf{X}, \mathsf{Y}) & \leftarrow \mathsf{father}(\mathsf{X}, \mathsf{Y}). \\
\mathsf{answer}(\mathsf{X}) & \leftarrow \mathsf{grandparent}(\mathsf{ann}, \mathsf{X}).
\end{array}$$

Here, father and mother are EDB-predicates, so they are defined by a collection of facts in a database. The initial state is shown in the following picture.



The links between rules correspond to the "is called by" relationship. Actually, one can start at a rule in the state and compose a goal in the SLD-tree by following the links and replacing the selected literal of the calling rule by the goal computed for the called rule (initially the body). Of course, a unification must be done, but it is guaranteed to succeed, because the constraints of the calling rule were propagated when the node for the called rule was constructed (e.g. the argument ann in the example). The advantage of the proposed method is that many nodes of the SLD-tree are encoded in a single state — in some cases infinitely many: E.g. $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{p}(\mathsf{X})$ gives an infinite SLD-tree, but in the state it is represented only once with a cyclic "is called by" relationship.

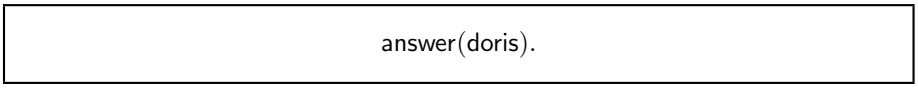
From one state to the next, a single EDB-fact is used. The distinction between program rules and EDB facts is important, because for the partial evaluation, we will later assume that the database is known only at runtime (and can change between one execution of the "compiled" program and the next). For example, using the fact $\mathsf{mother}(\mathsf{ann}, \mathsf{betty})$, we get the state shown in the next picture:



The proven literal is removed from the rule, and matching rules are found for the next selected literal.

In this example, backtracking will be needed, because another fact, e.g. $\mathsf{father}(\mathsf{ann}, \mathsf{chris})$, could be applied in the initial state. However, in many examples, especially for parsing applications, it can be proven that only a single EDB-fact can be applied, and therefore no backtracking will be necessary.

With a second state transition, an answer will be computed. For instance, with the fact $\mathsf{mother}(\mathsf{betty}, \mathsf{doris})$, we arrive at the following state:



The computed IDB-facts $\mathsf{parent}(\mathsf{betty}, \mathsf{doris})$ and $\mathsf{grandparent}(\mathsf{ann}, \mathsf{doris})$ were removed as part of the state cleanup operation (together with the rules which cannot yield further answers in a future state transition). □

Example 2. Let the left recursive version of the standard transitive closure program be given:

[1] $\mathsf{path}(\mathsf{X}, \mathsf{Y}) \leftarrow \mathsf{edge}(\mathsf{X}, \mathsf{Y})$.
[2] $\mathsf{path}(\mathsf{X}, \mathsf{Z}) \leftarrow \mathsf{path}(\mathsf{X}, \mathsf{Y}) \wedge \mathsf{edge}(\mathsf{Y}, \mathsf{Z})$.

Let the database be

[3] $\mathsf{edge}(1, 2)$.
[4] $\mathsf{edge}(2, 3)$.

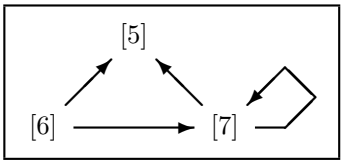
Now let the following query be given:

[5] $\mathsf{answer}(\mathsf{X}) \leftarrow \mathsf{path}(1, \mathsf{X})$.

The initial state $\mathcal{S}_0$ consists of the query plus rules added by instantiation:

[6] $\mathsf{path}(1, \mathsf{X}) \leftarrow \mathsf{edge}(1, \mathsf{X})$. // inst. of [1] because of [5]
[7] $\mathsf{path}(1, \mathsf{X}) \leftarrow \mathsf{path}(1, \mathsf{Y}) \wedge \mathsf{edge}(\mathsf{Y}, \mathsf{X})$. // inst. of [2] because of [5]

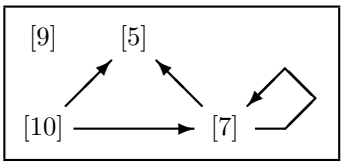
Of course, [7] also calls for instantiation, but that gives again [6] and [7]. Thus, $\mathcal{S}_0$ is



Now by applying the database fact [3] $\mathsf{edge}(1,2)$, the following reductions are done:

[8] $\mathsf{path}(1,2)$. // Reduction of [6] with [3]
[9] $\mathsf{answer}(2)$. // Reduction of [5] with [8]
[10] $\mathsf{path}(1,\mathsf{X}) \leftarrow \mathsf{edge}(2,\mathsf{X})$. // Reduction of [7] with [8]

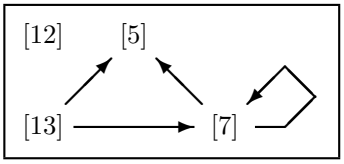
Standard IDB-facts like [8] are eliminated at the end of the construction of the successor state (with the cleanup operation). Answer facts like [9] are kept to show the answer(s) computed in the state, but they have no links (the answer-predicate cannot appear in rule bodies). The interesting case are rules which are not finished like [10]: Such rules remain in the state, and they have links to the same rules as the original rule, from which they were generated by reduction, so in this case it has links to [5] and to [7]. Therefore, these rules are copied to the successor state $\mathcal{S}_1$:



Next, using the database fact [4] $\mathsf{edge}(2,3)$ in state $\mathcal{S}_1$ permits the following reductions:

[11] $\mathsf{path}(1,3)$. // Reduction of [10] with [4]
[12] $\mathsf{answer}(3)$. // Reduction of [5] with [11]
[13] $\mathsf{path}(1,\mathsf{X}) \leftarrow \mathsf{edge}(3,\mathsf{X})$. // Reduction of [7] with [11]

As explained before, [11] is only temporarily needed while the reductions are done, [12] is the computed answer, and [13] has links to [5] and [7] (inherited from [7] from which it was constructed). Thus, the successor state $\mathcal{S}_2$ is:



Any other application of the database facts leads to the empty set (i.e. fails).

Note that $\mathcal{S}_1$ and $\mathcal{S}_2$ differ only in one constant (3 instead of 2). Partial evaluation, discussed below, will produce one "parameterized state", with this constant as a parameter. □

Theorem 1 (Correctness). *Every computed answer is indeed a logical consequence of* $\mathsf{P} \cup \mathsf{D}$.

Proof. This is easy: Each step (reduction and instantiation) is a logical consequence of $\mathsf{P} \cup \mathsf{D}$ and the previously computed rules. □

Definition 17 (Bottom-Up Derivation Tree). *Let a program* P *and database* D *be given. A bottom-up derivation tree is a finite tree with nodes labelled by facts, such that for each node the following holds: Suppose the node is labelled with* F. *Then either* $\mathsf{F} \in \mathsf{D}$, *and the node is a leaf node, or there is a rule* $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \in \mathsf{P}$ *and a ground substitution* θ *for this rule such that* $\mathsf{F} = \mathsf{A}\,\theta$ *and the child nodes are marked with* $\mathsf{B}_i\,\theta$ *for* $i = 1, \ldots, n$ *in this sequence. (This assumes the first literal selection rule, otherwise the subtrees must be re-ordered corresponding to the evaluation sequence of the body literals.) A bottom-up derivation tree for* F *is such a tree with the root node labelled with* F. *The EDB fact sequence of a bottom-up derivation tree is the sequence of labels of the leaf nodes (in the order defined by the tree).*

It is this fact sequence which must be used to get a state sequence which derives the fact at the root of the tree:

Theorem 2. *Let* $\mathcal{S}_0 = (\mathcal{V}_0, \mathcal{E}_0)$ *be a state containing a rule* $R = \mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n$ *with selected literal* B_1. *Let* θ *be a ground substitution for the variables in* B_1 *such that* $\mathsf{B}_1\,\theta$ *has a bottom-up derivation tree with EDB fact sequence* $\mathsf{F}_1, \ldots, \mathsf{F}_m$. *Then there are successor states* $\mathcal{S}_i$ *of* $\mathcal{S}_{i-1}$ *wrt* F_i, $i = 1, \ldots, m$, *such that the following holds:*

- *If* $n > 1$, *then* $(\mathsf{A} \leftarrow \mathsf{B}_2 \wedge \cdots \wedge \mathsf{B}_n)\,\theta$ *is contained in* $\mathcal{S}_m$. *Furthermore, every rule* $R' \in \mathcal{V}_0$ *which is reachable from* R *via* $\mathcal{E}_0$ *is still contained in every* $\mathcal{S}_i$, $i = 1, \ldots, m$, *as are the edges between these rules.*
- *If* $n = 1$, *then* $\mathsf{A}\,\theta$ *is generated by reduction from* R *with* F_m *in the last state transition (since it is an IDB fact, it will finally be removed, unless it is an* answer*-fact, but all further reductions with it are done). Furthermore, every rule* $R' \in \mathcal{V}_0$ *which is reachable from* R *via* $\mathcal{E}_0$ *is still contained in every* $\mathcal{S}_i$, $i = 1, \ldots, m-1$ *(i.e. until the penultimate state), as are the edges between these rules.*

Proof. The proof is by induction on the height of the bottom-up derivation tree for $\mathsf{B}_1\,\theta$. For space reasons, the details are contained only in the technical report version of this paper. □

Corollary 1 (Completeness). *Every fact* $\mathsf{answer}(c_1, \ldots, c_n)$ *which is a logical consequence of* $\mathsf{P} \cup \mathsf{D}$ *is computed.*

4 Partial Evaluation

The goal of partial evaluation is to do the main work of the computation of the states once and for all at "compile time", when the actual extensions of

the EDB-predicates are not yet known. Furthermore, even when only a single run of the program is planned, it can be advantageous to compute beforehand what can be done independently of actual constants. Our deduction method is intended for data-intensive applications, so there are relatively few rules and a large number of facts. Therefore, basically the same computation steps will be repeated many times for different data values (constants). Whatever can be done independently of the actual constant should be done only once. Indeed, only with partial evaluation our deduction method has a chance to be superior to other established methods.

Partial evaluation requires to separate what is known at compile time from what is known only later at runtime. In our case, the actual values of constants from EDB facts are not yet known. We represent them by special variables $C_1, C_2, \ldots$ and remember that they will actually be bound at runtime, so there will be a ground substitution given for them. Furthermore, the substitution is the same for the complete set of rules forming a "parameterized state". Therefore, these special variables are not local to the rules, but have a value globally defined for the entire state. At runtime, a state can be represented by the number of the parameterized state and the ground substitution for the special variables. If one wants to express the computation with states again as Datalog, one could create a predicate for each state with the parameters as arguments. This is useful for explaining the method (and for comparing our approach with Magic Sets). However, in the end, our goal is to translate directly to an implementation language like C++.

Example 3. Consider the following program:

[1] $\mathsf{answer}(\mathsf{X}) \leftarrow \mathsf{edge}(\mathsf{a}, \mathsf{X})$.
[2] $\mathsf{answer}(\mathsf{X}) \leftarrow \mathsf{edge}(\mathsf{b}, \mathsf{X})$.

This is also the initial state $\mathcal{S}_0$ (modulo rule normalization). It does not contain parameters: constants in the program are known at compile time. Now the successor state must be computed for the fact

[3] $\mathsf{edge}(C_1, C_2)$.

The actual constants are not yet known, therefore they are replaced by parameters C_1 and C_2. When reduction is applied to the first rule, we get the unifier $\theta_1 = \{C_1/\mathsf{a},\ \mathsf{X}/C_2\}$. An alternative would be $\theta_1' = \{C_1/\mathsf{a},\ C_2/\mathsf{X}\}$. However, since substitutions for the parameters must be treated specially, it is easier to orient variable-to-variable bindings in direction to the parameter.

Whenever a parameter gets bound to a constant, we must distinguish two cases: Either the actual value at runtime is equal to the constant, or not. We might get two different successor states in both cases, although often, there will be no successor state in the other case (i.e. we get the empty set of rules). Of course, the same happens when the substitution binds two parameters together, e.g. $\{C_1/C_2\}$. One case is that the two values are indeed equal, i.e. $C_1 = C_2$, the other is that they are not equal (in which case the unifier is not acceptable, and the result of the reduction or instantiation step is not included in the successor

state for this case). Of course, the conditions for a case must be consistent. If we assume that $C_1 \neq C_2$, we cannot bind C_1 and C_2 later to the same constant. This is a consequence of the fact that the parameters are like global variables for the state, so they cannot be replaced by different values for different rules in the state.

But let us continue with the example. The first case is $C_1 = \mathsf{a}$. Reduction gives the rule

[4] $\mathsf{answer}(C_2)$.

No further reduction can be applied under the assumption $C_1 = \mathsf{a}$, thus this is already one possible successor state. It contains a parameter, so we call it $\mathcal{S}_1(C_2)$. One can encode the state transition as the following rule:

[5] $\mathcal{S}_1(C_2) \leftarrow \mathcal{S}_0 \wedge \mathsf{edge}(C_1, C_2) \wedge C_1 = \mathsf{a}$.

The other case is $C_1 \neq \mathsf{a}$. Then it is clear that a reduction with the first rule is not possible. A reduction step with the second rule gives the substitution $\{C_1/\mathsf{b}, \mathsf{X}/C_2\}$ and the resulting state is

[6] $\mathsf{answer}(C_2)$.

This happens to be the same as the second state, so we encode it as $\mathcal{S}_1(C_2)$, too. The condition is $C_1 \neq \mathsf{a} \wedge C_1 = \mathsf{b}$. Of course, this could be simplified, since the second part implies the first one. But since it will be implemented as an "if ... else if ..."-chain, this simplification is not important. Finally, the case $C_1 \neq \mathsf{a} \wedge C_1 \neq \mathsf{b}$ remains. But then no reduction is possible, and thus, there is no successor state in this case. □

Example 4. This example explains why the negation of the condition of a substitution must be considered, too. Let the following program be given:

[1] $\mathsf{answer}(\mathsf{X}, 1) \leftarrow \mathsf{edge}(\mathsf{a}, \mathsf{X})$.
[2] $\mathsf{answer}(\mathsf{Y}, 2) \leftarrow \mathsf{edge}(\mathsf{X}, \mathsf{Y}) \wedge \mathsf{color}(\mathsf{X}, \mathsf{red})$.

Again, this is the initial state $\mathcal{S}_0$. The successor state must be computed for

[3] $\mathsf{edge}(C_1, C_2)$.

First, a reduction step with the first rule is possible and gives the substitution $\{C_1/\mathsf{a}, \mathsf{X}/C_2\}$. Therefore, we consider the case $C_1 = \mathsf{a}$. This needs to hold for the entire result state, so C_1 is replaced in all previously generated rules by a, and in all further derivation steps for this case. Therefore, the result state is

[4] $\mathsf{answer}(C_2, 1)$.
[5] $\mathsf{answer}(C_2, 2) \leftarrow \mathsf{color}(\mathsf{a}, \mathsf{red})$.

Let us encode this as $\mathcal{S}_1(C_2)$.

The other case is $C_1 \neq \mathsf{a}$. In that case, a reduction with the first rule is not possible, only with the second rule. The result is

[6] $\mathsf{answer}(C_2, 2) \leftarrow \mathsf{color}(C_1, \mathsf{red})$.

This state will be called $\mathcal{S}_2(C_1, C_2)$. It is important to exclude $C_1 = \mathsf{a}$ here, because otherwise the same rule instance would be applied twice. This would reduce the efficiency, and if duplicates were important, we would get the wrong number. Partial evaluation should not change the number of derivations that are possible. The state transition can be described by the following rules:

[7] $\mathcal{S}_1(C_2) \leftarrow \mathcal{S}_0 \wedge \mathsf{edge}(C_1, C_2) \wedge C_1 = \mathsf{a}$.
[8] $\mathcal{S}_2(C_1, C_2) \leftarrow \mathcal{S}_0 \wedge \mathsf{edge}(C_1, C_2) \wedge C_1 \neq \mathsf{a}$.

The condition $\mathcal{S}_0$ is actually not needed ("true"), since there is always an initial state. □

Definition 18 (Parameterized State). *A parameterized state with n parameters is a finite directed graph $(\mathcal{V}, \mathcal{E})$ where the vertices are rules which can contain the special variables $C_1, \ldots, C_n$, standard variables $\mathsf{X}_1, \mathsf{X}_2, \ldots$, and no other variables. The standard variables must be numbered in each rule in the order of occurrence.*

When a parameterized state is computed, it can be checked whether it differs only in the numbering of the parameters from a previously computed state. Then the earlier representation can be used.

The inititial parameterized state is equal to the initial state (it has no parameters). It would also be possible to allow queries with parameters — so that the compilation is done only once, and the query can be executed several times with different values for the parameters. But the same effect can be reached by creating a new EDB-predicate for the parameter values, and storing the concrete values in the database before the query is executed.

Now let a parameterized state $\mathcal{S}$ be given with n parameters $C_1, \ldots, C_n$. To compute the successor state for an EDB-predicate r of arity k, we use the "parameterized fact" $\mathsf{r}(C_{n+1}, \ldots, C_{n+k})$. Whenever a unification is done, we must keep in mind that the C_i represent unknown constants. Therefore, bindings between standard variables and parameters are done from the standard variable to the parameter. Basically, the unifying substitution is split in two parts, which can be applied one after the other: One part maps the standard variables in rules from state and program, and one part maps the parameters C_i.

The first part of the substitution can be applied at compile time as as usual. The second part yields a condition for the ground substitution that is defined by the data at runtime.

Definition 19 (Condition for a Substitution). *Let a substitution θ for the parameters be given. The corresponding condition is the conjunction of*

- *$C_i = c$ for each parameter C_i such that $\theta(C_i)$ is a constant c, and*
- *$C_i = C_j$ for each parameter C_i such that $\theta(C_i)$ is a parameter C_j with $j \neq i$.*

The empty conjunction (in case $\theta(C_i) = C_i$ for all i) is treated as true.

Conversely, given any consistent conjunction of such equations and negated formulas, the corresponding substitution maps a parameter C_i to

- *the constant c if $C_i = c$ is logically implied,*
- *the parameter C_j if $j > i$ is the greatest natural number such that $C_i = C_j$ is logically implied,*
- *C_i otherwise.*

During the computation of the successor state, one keeps a formula φ which describes the current case. It is a conjunction of such formulas describing parameter substitutions and their negations. Whenever a unification is done during the computation of the successor state, which computes the unifier θ, two cases are distinguished, which lead to different successor parameterized states (at compile time, one uses backtracking to follow both paths, at runtime the actual constant values choose one case):

- The unification will be successful also with the real parameter values, thus the formula φ' corresponding to θ is added to the current case, i.e. $\varphi := \varphi \wedge \varphi'$.
- The unification will not be successful, thus the negation of φ' will hold for the parameter values, i.e. $\varphi := \varphi \wedge \neg\varphi'$. Since this is the case where the unification will fail at runtime, the resulting rule is not added to the successor state computed at compile time, i.e. the instantiation or reduction is not done in this case.

Of course, if φ gets inconsistent, the current case is void and one can backtrack to compute the next successor state. If the domains are large enough, it is not difficult to check these formulas for consistency: One starts with one set for each parameter and for each constant. Whenever an equation (non-negated, of course) makes two parameters or a parameter and a constant equal, one merges the two sets. If a set contains two distinct constants, the formula is obviously inconsistent. After one is done with the non-negated equations, one checks each negated conjunction: If it contains at least one equation where the two sides belong to different sets, it is satisfied (in the interpretation which assigns every set a different value, which is the best possible interpretation for trying to satisfy inequalities, i.e. if there is one negated conjunction not satisfied in this interpretation, it will not be satisfied in any model of the positive part).

Now suppose one has computed a successor state and a formula φ describing the condition on the parameters. Because some unifications might have introduced additional constraints on the parameters which are not contained in other unifications, it is advantageous to apply the substitution corresponding to φ to all rules in the state. This is not strictly needed, since the parameter values must satisfy φ when the successor state is reached, but it might reduce the number of parameters in the state. Suppose that m parameters remain in the state. They are ordered in some sequence as $C_{i_1}, \ldots, C_{i_m}$ (trying to reuse already computed states if possible). Now in the state, the parameter C_{i_j} is replaced by C_j. Let the result be $\mathcal{S}'$. Then the state transition corresponds to the following rule:

$$\mathcal{S}'(C_{i_1}, \ldots, C_{i_m}) \leftarrow \mathcal{S}(C_1, \ldots, C_n) \wedge \mathsf{r}(C_{n+1}, \ldots, C_{n+k}) \wedge \varphi.$$

Example 5. In the grandparent example (Example 1), the result of partial evaluation looks as follows:

$$
\begin{array}{ll}
\mathcal{S}_0. & \\
\mathcal{S}_1(C_2) & \leftarrow \mathcal{S}_0 \wedge \mathsf{mother}(C_1, C_2) \wedge C_1 = \mathsf{ann}. \\
\mathcal{S}_1(C_2) & \leftarrow \mathcal{S}_0 \wedge \mathsf{father}(C_1, C_2) \wedge C_1 = \mathsf{ann}. \\
\mathcal{S}_2(C_3) & \leftarrow \mathcal{S}_1(C_1) \wedge \mathsf{mother}(C_2, C_3) \wedge C_1 = C_2. \\
\mathcal{S}_2(C_3) & \leftarrow \mathcal{S}_1(C_1) \wedge \mathsf{father}(C_2, C_3) \wedge C_1 = C_2. \\
\mathsf{answer}(C_1) & \leftarrow \mathcal{S}_2(C_1).
\end{array}
$$

Of course, one could optimize the initial state $\mathcal{S}_0$ away, and apply the equations to reduce the number of parameters. For the planned translation to C++, this is not important, but for a comparison with magic sets, one would do this. In the grandparent example, the result of the magic set transformation is recursive. There are solutions for this, but magic sets is far from using one EDB fact in each rule application. If one compares the number of applicable rule instances (which does not count the recursion overhead), magic sets needs 18 compared to our 10. More arguments in derived predicates and a larger join is needed. □

5 Termination of Partial Evaluation

In general, it is possible that partial evaluation does not terminate, i.e. it generates states with more and more parameters. While a concrete database state has only a fixed number of data values, the result of partial evaluation must work for a state of any size. There are two solutions to this problem:

- One can find syntactic conditions for programs where it is guaranteed that partial evaluation terminates. An easy case are non-recursive programs, but we strongly presume that this holds also for left-recursive programs.
  It seems that right-recursive programs can also be allowed, if one replaces instantiation and reduction on the last literal of a rule (if it is an IDB-literal) by a single resolution step as in SLD-resolution ("last literal resolution").
- While left- and right-recursion are sufficient for many interesting applications, and deserve an especially efficient implementation, the relation to parsing shows that it cannot always work this way: Basically, what we have constructed is a finite state machine which walks over the EDB facts. The states have parameters, but the amount of information stored in them is bounded. We need backtracking only when different facts can be applied in a state, but this does not happen for parsing applications. Therefore, it is obvious that there are cases where a stack is needed for computing a single solution. A trick already used for SLDMagic [2] works here, too: For a recursive call, which is neither left- nor right-recursive, one can set up a subquery, and later treat the solution like an EDB-fact. Another possible solution, more in the spirit of Earleys algorithm, is not to copy all reachable rules from one state to the next, but to set up "global links" in such cases.

6 Conclusions

The proposed method (with partial evaluation) saves work at runtime in two cases: First, long chains of calls between IDB-predicates are optimized away,

since in every step an EDB-fact is used. The number of leaf nodes in a bottom-up derivation tree for an answer is the number of state transitions needed to compute this answer. Second, if there are several rules called in the same state and having a common prefix in the body, this prefix is processed in parallel.

Furthermore, it is easily visible which EDB-facts could be used in a given state. Often, it is clear that there can be only one applicable fact, e.g. because of key constraints or mutual exclusions. Then no preparation for backtracking is needed, and it might be possible to reuse the storage space for the state, and to avoid copying operations for the parameters.

Progress is reported at: `http://dbs.informatik.uni-halle.de/Earley`.

References

1. Angeles, R., Gutierrez: The expressive power of SPARQL. In: Sheth, A., et al. (eds.) The 7th International Semantic Web Conference (ISWC 2008). pp. 114–129. No. 5318 in LNCS, Springer (2008), `http://www.dcc.uchile.cl/cgutierr/papers/expPowSPARQL.pdf`
2. Brass, S.: SLDMagic — the real magic (with applications to web queries). In: Lloyd, W., et al. (eds.) First International Conference on Computational Logic (CL'2000/DOOD'2000). pp. 1063–1077. No. 1861 in LNCS, Springer, Heidelberg, Berlin (2000)
3. Brass, S.: Implementation alternatives for bottom-up evaluation. In: Hermenegildo, M., Schaub, T. (eds.) Technical Communications of the 26th International Conference on Logic Programming (ICLP'10). Leibniz International Proceedings in Informatics (LIPIcs), vol. 7, pp. 44–53. Schloss Dagstuhl (2010), `http://drops.dagstuhl.de/opus/volltexte/2010/2582`
4. Brass, S.: Order in datalog with applications to declarative output. In: Barceló, P., Pichler, R. (eds.) Datalog in Academica and Industry, 2nd Int. Workshop, Datalog 2.0. LNCS, vol. 7494, pp. 56–67. Springer-Verlag (2012), `http://users.informatik.uni-halle.de/~brass/order/`
5. Bry, F., Furche, T., Ley, C., Marnette, B., Linse, B., Schaffert, S.: Datalog relaunched: Simulation unification and value invention. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog Reloaded, First International Workshop, Datalog 2010. pp. 321–350. No. 6702 in LNCS, Springer-Verlag (2011)
6. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. In: Proc. of the 28th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'09). pp. 77–86. ACM (2009)
7. Pereira, F.C.N., Warren, D.H.D.: Parsing as deduction. In: Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics (ACL). pp. 137–144 (1983), `http://www.aclweb.org/anthology/P83-1021`
8. Polleres, A.: From SPARQL to rules (and back). In: Proceedings of the Sixteenth International World Wide Web Conference (WWW2007). pp. 787–796 (2007), `http://wwwconference.org/www2007/papers/paper435.pdf`
9. Polleres, A.: How (well) do Datalog, SPARQL and RIF interplay? In: Barceló, P., Pichler, R. (eds.) Datalog in Academica and Industry, 2nd Int. Workshop, Datalog 2.0. pp. 27–30. No. 7494 in LNCS, Springer (2012)
10. Porter III, H.H.: Earley deduction (1986), `http://web.cecs.pdx.edu/~harry/earley/earley.pdf`