TM-0611

On a Recursive Query Processing Method for Deductive Databases

by
H. Seki

October, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

On A Recursive Query Processing Method for Deductive Databases

(Extended Abstract)

Hirohisa SEKI

Institute for New Generation Computer Technology

1-4-28, Mita, Minato-ku, Tokyo 108, Japan

1 Introduction

Recursive query processing in deductive databases has attracted much attention recently, and both top-down and bottom-up algorithms for query evaluation have been proposed by many authors ([1], [11] and references therein). This paper gives a bottom-up query evaluation algorithm called *Alexander Templates (AT)*, particularly emphasizing comparison with a top-down evaluation with memo-ization such as the SLD-AL procedure [16] (or QSQ [18]) and Extension Tables [6].

AT is a generalization of the Alexander Method [12], and it is applicable to general Horn programs, allowing partially bound terms (such as $p(f(a, X), Y)$) and repeated variables (such as $p(X, X)$). The basic principle of AT are similar to those of (Generalized) Magic Sets [5], [3], that is, a program is transformed into a set of rules whose bottom-up evaluation is devised to simulate top-down evaluation of the original program. We show that there exists an exact one-to-one correspondence between the bottom-up evaluation of the transformed rules and the top-down evaluation of the original program by the SLD-AL procedure. Thus, the bottom-up computation can be considered to have at least the same power as the top-down with memo-ization.

Once the correspondence between AT and the top-down evaluation with memo-ization is established, properties of AT such as correctness, complexity and termination are immediately derived from the counterparts of the SLD-AL procedure. For example, the bottom-up evaluation of AT is shown to be sip-optimal [11] (except supplementary atoms). Moreover, under the same sip and control strategy, the counts of generation of facts and solutions are shown to be equivalent in both AT and the SLD-AL procedure. We also propose a technique which improves the termination property of the bottom-up evaluation of AT, which is also applicable to (Generalized) Magic Sets.

Since the completion of the first version of this paper [13], we have recently found similar work by Ramakrishnan [11]. He proposed the *Magic Templates (MT)*, which is a generalization of Generalized Magic Sets [5] in the same sense that our AT is a generalization of the Alexander method (thus, we borrow the name *template* for our method). AT is similar to a *supplementary* version of MT (SMT).

It is, however, further optimized by applying unfolding to rules of SMT, which reduces intermediately generated supplementary atoms during the bottom-up evaluation. Furthermore, in order to show the correspondence between bottom-up and top-down with memo-ization, AT gives a nicer framework than MT.

The organization of this paper is as follows. After summarizing preliminary terminologies, section 2 describes Alexander Templates algorithm. Section 3 gives a brief explanation of a top-down evaluation method with memo-ization, the SLD-AL procedure. Section 4 gives the main result of this paper, which shows the one-to-one correspondence between the behaviour of AT and that of the SLD-AL procedure. Several corollaries on the properties such as termination, sip-optimality and termination, are also given. Section 5 describes termination issues of AT in more detail and proposes a technique to guarantee its termination for a more general class.

2 Alexander Templates

2.1 Preliminaries

We consider general Horn clauses (with function symbols) as a database. We mostly follow basic terminologies and conventions given in [11] and [4]. Throughout this paper, P means a program consisting of a finite set of Horn clauses (rules), and D a database which is a set of facts (i.e., not necessarily ground unit clauses). Predicates in D are called *database* predicates, while all other predicates are called *derived*. No database predicate is assumed to appear in the head of a rule in P. Furthermore, each rule in P is supposed to be assigned a unique natural number called *rule number*. We denote a (possibly empty) set of all the variables occurring in an atom, A, by $var(A)$. Similarly, $var(A_1 \cup ... \cup A_n)$ means a set of all the variables occurring among atoms $A_1, ..., A_n$. For simplicity, we denote this set by a list form. For example, $var(p(X,Y))$ is denoted by $[X,Y]$.

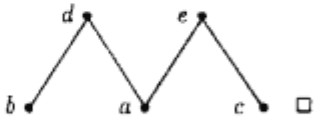
Example 2.1 Consider the following program P_{sg} of the same generation problem [4].

$$sg(X,Y) \leftarrow flat(X,Y). \quad (1)$$

$$sg(X,Y) \leftarrow up(X,Z1), sg(Z1,Z2), flat(Z2,Z3), sg(Z3,Z4), down(Z4,Y). \quad (2)$$

To each rule (1) ((2)), the number in the parentheses is assumed to be assigned as its rule number, respectively. We also assume the following database D_{sg} is given.

$$\begin{array}{lllll} flat(a,a). & flat(b,b). & flat(c,c). & flat(d,d). & flat(e,e). \\ up(a,d). & up(a,e). & up(b,d). & up(c,e). & \\ down(d,a). & down(e,a). & down(d,b). & down(e,c). & \end{array}$$



□

In Magic Templates and also in AT, a given program P is transformed into a new set of (adorned) rules P^{ad} based on a *sideways information passing* (sip) [11], [5]. For simplicity of exposition, we assume throughout this paper that a *full* sip (see [11]) is used. Thus, there exists a *total* order in all atoms of

each rule in P, and we can assume that the body literals of each rule in P are ordered according to its total ordering induced by the total sip. Then, as is described in [11], P and P^{ad} are identical, with the understanding that in P^{ad}, each n-ary predicate is adorned with a string of n *bs*. In the following, we thus use P in stead of P^{ad}

2.2 CP Rule Transformation

Our rule transformation of AT is a generalization of the Alexander method, in the same sense that MT is a generalization of Generalized Magic Sets [5]. It is applicable to general Horn programs. Rules of a given program are transformed into a set of new rules which we call *continuation passing (CP) rules*.

An informal explanation of CP rules is as follows. Consider a rule r of form: "$p \leftarrow \Gamma, q, \Lambda$", where Γ and Λ are (possibly empty) sequences of atoms and q is an arbitrary atom. This rule could be divided into the following two rules:

$$\begin{aligned} call\_q,\ cont\_q &\leftarrow call\_p, \Gamma && \cdots(*1) \\ sol\_p &\leftarrow sol\_q,\ cont\_q,\ \Lambda && \cdots(*2) \end{aligned}$$

The first rule (*1) is simply a convention of two rules: "$call\_q \leftarrow call\_p, \Gamma$" and "$cont\_q \leftarrow call\_p, \Gamma$". The atom $call\_q$ plays the same role with that of $magic\_q$ in Magic Sets. The intention of rule (*1) is that, if a query of p is given (this is the meaning of $call\_p$), and subgoals Γ are solved, then a subgoal, q, is to be tried next (this is the meaning of $call\_q$). The information obtained during solving subgoals Γ is stored in an atom, $cont\_q$. On the other hand, the second rule (*2) means that, if a goal q is solved (a solution of q is denoted by sol_q), then, an attempt is made to solve subgoal Λ under the constraint specified by $cont\_q$. If Λ is solved, then a solution of p is obtained in the form sol_p. Thus, forward reasoning interpretation of the original rule r could be considered to be rewritten into rules (*1) and (*2).

Before describing the transformation algorithm from a given program into CP rules, we introduce the following definitions.

Definition 2.1 An atom which has a prefix, $call\_$ ($sol\_$), is said to be a *call-atom (sol-atom)*. An atom whose predicate symbol is *cont* is said to be a *cont-atom* (thus, "*cont*" is a reserved predicate). These call-atoms, sol-atoms and cont-atoms are called *CP-atoms*.

A rule is said to be a *continuation passing (CP) rule* if its body contains only CP-atoms.

Let $\leftarrow q$ be a given query. The call-atom of $call\_q$ is said to be an *initial call-atom for* $\leftarrow q$. □

For example, let $\leftarrow sg(a, Y)$ be a query. Then, its initial call-atom is $call\_sg(a, Y)$.

The algorithm for transforming rules in P and facts in D into the corresponding *CP rules* consists of the following four steps.

Transformation Algorithm:

Input : a given program P, a database D and a query Q

Output : a set of CP rules, $CPRs(P \cup D)$, and initial call-atom $call\_Q$

Procedure :

Let r be an arbitrary rule in P or D of form: $A \leftarrow A_1, ..., A_n (n \geq 0)$. When r is in P, let N_r be its rule number. Let *CPRs(r)* be an empty set. Apply the following steps to r.

step 1 : {Introduction of call/sol-predicates} For a rule, r, the following rule, r_{new}, is introduced:

$$sol\_A \leftarrow call\_A, A_1, ..., A_n \tag{3}$$

step 2 : {Termination condition} If rule r_{new} is already a CP rule, then let *CPRs(r)* be *CPRs(r)* $\cup\{r_{new}\}$, and terminates the process of deriving CP rules for r. Otherwise, apply the following **step 3** to r_{new}.

step 3 : {Cutting off recursion} Let A_k be the leftmost atom in the body of r_{new} such that it is not a CP-atom. Suppose that A_k is the k-th ($1 \leq k \leq n$) atom in the body of the original rule r. Note that r_{new} is of the following form:

$$sol\_A \leftarrow \Gamma, A_k, \Lambda \tag{4}$$

where Γ is a sequence consisting only of CP-atoms, and Λ is a (possibly empty) sequence of atoms. Then, we rewrite r_{new} into the following two rules:

$$call\_A_k, cont(N_r\text{-}k, IV_k, CV_k) \leftarrow \Gamma \tag{5}$$

$$sol\_A \leftarrow sol\_A_k, cont(N_r\text{-}k, IV_k, CV_k), \Lambda \tag{6}$$

where IV_k (called *internal variables* of A_k) means a set of variables (if any) in A_k in (4), and CV_k (called *continuation variables*) means a set of all the variables in rule (4) which occur among both "Γ" and "Λ, sol_A" simultaneously but not in A_k. Namely,

$$CV_k = var(\Gamma) \cap var(\Lambda \cup sol\_A) - IV_k$$

Each of IV_k and CV_k is denoted in list form. Each variable in lists IV_k and CV_k is ordered according to the order of its textual appearance in rule r_{new}. Since rule (5) is a CP rule, let *CPRs(r)* be *CPRs(r)* $\cup\{(5)\}$.

step 4 : {Applying step 2 and step 3 until the end of the rule body} Let (6) be a new r_{new}, and go back to **step 2**.

Note that the above process always terminates. We apply this procedure to every rule r in $P \cup D$. Then, the output of the transformation algorithm is $\bigcup_{r \in P \cup D} CPRs(r)$. □

Example 2.2 Continued from **Example 2.1**. The following is the set of derived CP rules $CPRs(P_{sg} \cup D_{sg})$ (a detailed process of transformation is given in **Appendix-A1**):

$$call\_flat(X, Y), cont(1\text{-}1, [X, Y], [\,]) \leftarrow call\_sg(X, Y) \tag{7}$$

$$sol\_sg(X, Y) \leftarrow sol\_flat(X, Y), cont(1\text{-}1, [X, Y], [\,]) \tag{8}$$

$$call\_up(X, Z1), cont(2\text{-}1, [X, Z1], [Y]) \leftarrow call\_sg(X, Y) \tag{9}$$

$$call\_sg(Z1, Z2), cont(2\text{-}2, [Z1, Z2], [X, Y]) \leftarrow sol\_up(X, Z1), cont(2\text{-}1, [X, Z1], [Y]) \tag{10}$$

$$call\_flat(Z2, Z3), cont(2\text{-}3, [Z2, Z3], [X, Y]) \leftarrow sol\_sg(Z1, Z2), cont(2\text{-}2, [Z1, Z2], [X, Y]) \quad (11)$$

$$call\_sg(Z3, Z4), cont(2\text{-}4, [Z3, Z4], [X, Y]) \leftarrow sol\_flat(Z2, Z3), cont(2\text{-}3, [Z2, Z3], [X, Y]) \quad (12)$$

$$call\_down(Z4, Y), cont(2\text{-}5, [Z4, Y], [X]) \leftarrow sol\_sg(Z3, Z4), cont(2\text{-}4, [Z3, Z4], [X, Y]) \quad (13)$$

$$sol\_sg(X, Y) \leftarrow sol\_down(Z4, Y), cont(2\text{-}5, [Z4, Y], [X]) \quad (14)$$

$$sol\_flat(a, a) \leftarrow call\_flat(a, a)$$

$$\cdots$$

$$sol\_down(e, c) \leftarrow call\_down(e, c) \qquad \square$$

2.2.1 Notes on the rule transformation

For a given query $\leftarrow Q$, it is obviously unnecessary to transform all rules and facts in $P \cup D$, but it is enough to transform only those related to the query. Furthermore, it might be curious that even facts in D are transformed into its corresponding CP rule, since we do not transform facts in D in usual query evaluation methods such as (Generalized) Magic Sets and MT. Of course, we could do such a rule transformation which does not modify predicates in D. We give such an example in **Appendix-A2**. In this paper, however, we adopt the above transformation for the ease of explanation of comparison with the corresponding top-down algorithm described in the following section.

For readers who are familiar with the supplementary version of Magic Templates or Generalized Supplementary Magic Sets (GSMS), we give a comparison with AT and SMT in **Appendix-A2** for the same generation problem. In that problem, $CPRs(P_{sg})$ is similar to a set $P^{sup\text{-}ms}$ of magic rules and modified rules rewritten by SMT, but $CPRs(P_{sg})$ is an further optimized result of $P^{sup\text{-}ms}$, by applying straightforward *unfolding* to it. Thus, the size of intermediate atoms generalized by bottom-up computation in AT becomes smaller than that of Supplementary MT or GSMS.

2.3 Naive Evaluation of CP Rules

Since a sip and a control strategy are quite another issues (see [5]), we can consider various control strategies for our CP rules. In this paper, we employ a naive bottom-up evaluation [4] as the control strategy.[1]. For a given set of CP rules $CPRs(P \cup D)$ and an initial call-atom $call\_q$, we compute a sequence of sets of atoms $S_0, \cdots, S_n, \cdots$ as follows:

$$S_0 := \{call\_q\} \quad (15)$$

$$S_{n+1} := T(S_n) \quad (16)$$

$$\equiv \{A\theta \mid A \leftarrow \Gamma \text{ is a rule in } CPRs(P \cup D),\ \theta \text{ is a substitution such that each atom in } \Gamma\theta \text{ is in } S_n\} \quad (17)$$

where variables (if any) in $A\theta$ in (17) are considered to be freshly introduced variables such that they are distinct from those in S_n. Note that, since each atom is no longer a ground atom, the evaluation of

[1] Of course, other optimizing strategies such as semi-naive evaluation are possible.

T in (16) is not a usual "join" operation, but a unification operation. The above evaluation terminates, when no *new* atoms are generated for some n. Here, an atom A in S_n is said to be *new* if A is not an instance of any atom in S_{n-1}. The solution for a given initial atom $call\_q$ is given by set $\{sol\_q' \mid sol\_q'$ is in $S(n)$ and q' is an instance of $q\}$.

Example 2.3 Continued from **Example 2.2**. The computation process of the naive evaluation for input $(CPRs(P_{sg} \cup D_{sg}), call\_sg(a, Y))$ is shown in **Appendix-A3** (Figure 1).

The correctness of AT and other properties are given in section 4.

3 Top-down Evaluation with Memo-ization

As a top-down method for query evaluation, we adopt the SLD-AL procedure [16] (or QSQ [18]) introduced by Vieille. The basic principle of SLD-AL is to prevent the interpreter from repeatedly trying to solve the "similar" goal and thus to cut off an infinite branch, by introducing memo-ization into SLD derivation (e.g., [10]). Similar work has been proposed by several researchers (e.g., [15] [6]). The following description of QSQ (or SLD-AL) is mainly borrowed from [8] and [16].

Definition 3.1 A *lemma* is an atom for a predicate in $P \cup D$. □

A computation rule [10] determines which literal in a goal is selected. We assume that our computation rule always selects the *leftmost* atom in a goal, as in usual Prolog interpreter. This corresponds to our assumption that a total sip is adopted in the previous section. Next, we define a precedence relationship between nodes in a tree, which specifies a search strategy in a derivation tree.

Definition 3.2 Node M *precedes* node N in a tree T if either M is an ancestor of N or M is visited before N in a preorder traversal of T. □

Definition 3.3 The leftmost literal L in a goal G of an SLD-AL tree (defined below) is *admissible* if there is no goal G' (its leftmost atom L') of T such that G' precedes G and L is an instance of L'. □

The SLD-AL tree is defined as follows.

Definition 3.4 An *SLD-AL tree* for query $\leftarrow Q$ with respect to (P, D) and Σ (a set of lemmas) is a tree T of goals such that :

(1) The root of T is $\leftarrow Q$.

(2) Nodes that are the empty goal have no children (success nodes).

(3) {*OLD extension*} Let $N =$ " $\leftarrow L_1, \Gamma$" be a non-empty goal of T and Γ a (possibly empty) sequence consisting of atoms and "call-exit markers" (introduced below). Suppose that L_1 is admissible. Let $C_1, \cdots, C_k (k \geq 0)$ be all clauses (if any) in (P, D) such that L_1 and C_i is resolvable (with mgu θ_i), and let C_i be of form: $A_i \leftarrow A_{1i}, \cdots A_{li} (li \geq 0)$. Then, add k child

nodes N_1,...,N_k to N, where each N_i is of form: $\leftarrow (A_{1i}, \cdots A_{li})\theta_i, [\![L_1\theta_i]\!], \Gamma\theta_i$. Each $[\![L_1\theta_i]\!]$ is called a *call-exit marker*[7].[2]

(4) {*Lemma extension*} If L_1 in $N =$ " $\leftarrow L_1, \Gamma$" is not admissible, then, for each lemma L in Σ that is unifiable with L_1 (let θ be its mgu), N has a child $\leftarrow \Gamma\theta$.

(5) {*Detecting Lemmas*} If node N is of form: " $\leftarrow [\![L]\!], \Gamma$" , then, N has a child $\leftarrow \Gamma$. □

Hence, an SLD-AL tree is an SLD tree except that an admissibility test and lemma extension are introduced. The following definition shows a role of the call-exit marker.

Definition 3.5 Let $G_i =$ " $\leftarrow L_i, \Gamma', G_{i+1}, \cdots, G_j$ be a derivation BR with mgu's θ_{i+1},...,θ_j in an SLD-AL tree. Suppose that G_j is of form: " $\leftarrow [\![L_1\theta_{i+1} \circ \cdots \circ \theta_j]\!], \Gamma\theta_{i+1} \circ \cdots \circ \theta_j$" and is the first goal in BR not containing any descendants of L_1. Then, the derivation BR is called a *proof segment* for L_1 starting at G_i proving $L_1\theta_{i+1} \cdots \theta_j$. $L_1\theta_{i+1} \cdots \theta_j$ is the lemma *corresponding to* the proof segment. □

Finally, the SLD-AL procedure is defined as follows. The following procedure is based on a breadth first strategy such that, at n-th ($n \geq 0$) iteration step, the construction of the SLD-AL tree is limited up to depth n.

Definition 3.6 The SLD-AL procedure for evaluating $\leftarrow Q$ with respect to (P, D) is as follows:

$n := 0$; $\Sigma_n := \phi$; Let T_0 be an SLD-tree consisting only of root $\leftarrow Q$;
repeat
  n := n+1;
  Construct an SLD-AL tree T_n for $\leftarrow Q$ wrt (P, D) and Σ_{n-1} such that the depth of T_n is n;
  Let $\Delta\Sigma$ be the set of lemmas for predicates corresponding to proof segments in Σ_n;
  $\Sigma_n := \Sigma_{n-1} \cup \Delta\Sigma$;
until $\Delta\Sigma \subseteq \Sigma_{n-1}$; Return all computed answers for $\leftarrow Q$ in T_n. □

The soundness and completeness of SLD-AL procedure is given by [16].

Example 3.1 Continued from **Example 2.1**. The SLD-AL procedure for $\leftarrow sg(a, Y)$ with respect to (P_{sg}, D_{sg}) is given in **Appendix-A4** (Figure 2).

4 Simulation of Top-down with Memo by AT

The following proposition establishes the one-to-one correspondence between the behavior of the SLD-AL procedure and that of AT.

[2] The intention of introducing a call-exit marker is to detect and store efficiently lemmas generated intermediately during the SLD-AL procedure. See **Definition 3.5**.

Proposition 4.1 Let P (D) be a given Horn program (database), respectively and $\leftarrow Q$ a given query. Consider the SLD-AL procedure for $\leftarrow Q$ wrt (P, D), and the naive evaluation of CP rules $CPRs(P \cup D)$ together with the initial call-atom *call_Q*. Then, the following correspondence between these two algorithms holds (in the below, S_n is defined in (15) and (16), while T_n is defined in **Definition 3.6**):

(i) *(initial stage)* $\leftarrow Q$ is the root of SLD-AL tree T_0, while, in the naive evaluation, (a possibly variant of) an atom of the form: *call_Q* is in S_0.

(ii) A node of form: "$\leftarrow A, \Gamma$" is derived in SLD-AL tree T_n ($n > 0$), if and only if (a variant of) an atom of the form: *call_A* is in S_n of the naive evaluation. Furthermore, A is admissible if and only if *call_A* is newly derived at n-th step in the naive evaluation.

(iii) A lemma L (or equivalently, a node of form: "$\leftarrow [L], \Gamma$") is derived in SLD-AL tree T_n ($n > 0$), if and only if (a variant of) an atom of the form: *sol_L* is in S_n of the naive evaluation. Furthermore, L is a newly derived lemma at n-th step in the SLD AL procedure, if and only if *sol_L* is newly derived at n-th step in the naive evaluation. □

The proof is done by induction on n, and it is found in the full paper [13]. The above proposition shows that, there exists an exact one-to-one correspondence between each node in an SLD-AL tree constructed by the SLD-AL procedure and an atom generated in the naive evaluation of CP rules. Once the above correspondence is established, the following properties of Alexander Template are immediate. At first, AT is sound and complete.

Corollary 4.1

(Soundness) If an atom *sol_A* is derived by the naive evaluation of $CPRs(P \cup D)$, then A is a logical consequence of $P \cup D$.

(Completeness) If an atom $A\theta$ is a logical consequence of (P, D), then a sol-atom *sol_*A_1 is derived by the naive evaluation of $CPRs(P \cup D)$ with an initial call-atom *call_A*, where A_1 subsumes $A\theta$. □

Next, we discuss the *sip-optimality* [11] of AT. In [11], Ramakrishnan gives definitions of a *sip-strategy* and sip-optimality as follows. Taken as input a query and a program together with a collection of sips, *sip-strategy* computes answers to the query under such a computation rule that it satisfies the following two conditions:

(1) If $p(\theta)$ is a query, and $p(\phi)$ is an answer, then $p(\phi)$ is computed.

(2) If $p(\theta)$ is a query, then for very rule with head predicate p, a query is constructed for every predicate in the rule body according to the sip for their rule.

Then, a *sip-optimal* strategy is defined to be a sip-strategy that generates only the facts and the queries required by the above definition for the predicates in the program. Sip-optimality is a natural concept to capture the properties of many previously proposed top-down/bottom-up methods for general Horn programs (see [4], [11] and references therein). As mentioned in [11], however, sip-optimality does not imply that facts and queries are not generated more than once.

Throughout this paper, we are confined into full sips. Under a full sip strategy, it is easy to see from the definition that the SLD-AL procedure is sip-optimal. Thus, again from **Proposition 4.1**, AT is shown to be sip-optimal, except supplementary cont-atoms generated. Furthermore, we can say more on the counts of generation of facts and queries in AT.

Corollary 4.2 The naive evaluation of $CPRs(P \cup D)$ is sip-optimal, except cont-atoms generated intermediately. Furthermore, the numbers of times facts and queries are generated in the naive evaluation of $CPRs(P \cup D)$ are the same as those in the SLD-AL procedure for $(P \cup D)$. □

The next corollary is concerned with the termination of the naive evaluation of CP rules.

Corollary 4.3 The naive evaluation of $CPRs(P \cup D)$, together with initial atom $call\_A$, terminates if and only if the SLD-AL procedure for $\leftarrow A$ with respect to (P, D) terminates.

In particular, when P is a *datalog* database, the naive evaluation of $CPRs(P \cup D)$ always terminates for any initial call-atom. □

5 More on Termination Issues

We show in the previous section that the naive evaluation of Alexander Templates faithfully simulates the behavior of top-down computation with memo-ization of the SLD-AL procedure, and the termination condition of both methods is thus the same.

Interestingly, AT (also MT) sometimes simulates the top-down with memo-ization too faithfully, and there exist programs P such that the naive evaluation of $CPRs(P \cup D)$ will not terminate even if the naive evaluation of (P, D) terminates. Consider the following example:

Example 5.1 Suppose that the following program P_{loop} is given (let D be empty).

$$leq\_two(s(s(0))).$$
$$leq\_two(X) \leftarrow leq\_two(s(X)).$$

Predicate $leq\_two(X)$ is supposed to hold if X is less than or equal to two. $CPRs(P_{loop})$ is as follows, where N is the rule number of the second rule in P_{loop}.

$$sol\_leq\_two(s(s(0))) \leftarrow call\_leq\_two(s(s(0)))$$
$$call\_leq\_two(s(X)), cont(N\text{-}1, [X], [\,]) \leftarrow call\_leq\_two(X).$$
$$sol\_leq\_two(X) \leftarrow sol\_leq\_two(s(X)), cont(N\text{-}1, [X], [\,])$$

The naive evaluation of P_{loop} terminates, generating atoms $\{leq\_two(0), leq\_two(s(0)), leq\_two(s(s(0)))\}$, as is expected. Suppose that query $\leftarrow leq\_two(s(s(s(0))))$ is given. The naive evaluation of $CPRs(P_{loop})$ together with initial call-atom $call\_leq\_two(s(s(s(0))))$ does not terminate, since it can repeatedly produce new atoms $call\_leq\_two(s(s(s(s(0)))))$, $call\_leq\_two(s(s(s(s(s(0))))))$, $\cdots$. In this case, there is no way to stop the computation, since newly generated atoms are not instances of previous ones. Note that the

same thing also holds in Magic Templates (see the discussion in [11]) and in the SLD-AL procedure. □

We propose a method which guarantees the termination of the naive evaluation of CP rules, at least when the corresponding evaluation of the original program terminate. In order to prevent the above infinite generation of distinct (call-)atoms, we introduce *term-depth abstraction* [15].

Definition 5.1 Let A be an atom of a predicate p, and k the term-depth of p[3]. Then, the *term-depth abstraction* of A, denoted by $abs(A)$, is A with every subterm of depth more than k replaced by distinct new variables. Let $\Gamma = A_1, \cdots, A_n$ be a sequence of atoms. Then, the term-depth abstraction wrt call-atom of Γ, denoted by $call\text{-}abs(\Gamma)$, is Γ with every call-atom A_i (if any) in it replaced by $abs(A_i)$. □

For example, if the term-depth of p is 1, $abs(p(s(s(X))))$ is $p(s(X))$. The following is a naive evaluation given in **2.3**, incorporating the term-depth abstraction into it.

$$S_0^{tda} := \{abs(call\_q)\} \tag{18}$$

$$S_{n+1}^{tda} := T_{tda}(S_n^{tda}) \tag{19}$$

$$\equiv \{call\text{-}abs(\Lambda\theta) \mid \Lambda \leftarrow \Gamma \text{ is a rule in } CPRs(P \cup D),\ \theta \text{ is a substitution such that each atom in } \Gamma\theta \text{ is in } S_n^{tda}\} \tag{20}$$

Example 5.2 Continued from **Example 5.1**. Let the term-depth of *leq_two* (thus, *call_leq_two*) be 3. Then, evaluation of $CPRs(P_{loop})$ together with initial call-atom $call\_leq\_two(s(s(s(0))))$ terminates. The following is a sequence of sets of atoms generated at each iteration step.

$$\begin{aligned}
S_0^{tda} &= \{call_leq_two(s(s(s(0))))\} \\
S_1^{tda} &= S_0^{tda} \cup \{call_leq_two(s(s(s(X)))), cont(N\text{-}1, [X], [\,])\} \\
S_2^{tda} &= S_1^{tda} \cup \{call_leq_two(s(s(s(Y)))), cont(N\text{-}1, [s(s(s(Z)))], [\,])\} \\
S_{n+1}^{tda} &= S_n^{tda} \qquad (n \geq 2)
\end{aligned}$$

Note that each newly generated atom in S_2^{tda}, namely, $call\_leq\_two(s(s(s(Y))))$ and $cont(N\text{-}1, [s(s(s(Z)))], [\,])$, is an instance of the call-atom and the cont-atom in S_1^{tda}, respectively. Thus, the above naive evaluation with term-depth abstraction terminates at step 2. □

The following property on the termination of the above evaluation is easily shown [13].

Corollary 5.1 The naive evaluation with term-depth abstraction of $CPRs(P \cup D)$, together with initial call-atom $call\_Q$, terminates, whenever the naive evaluation of P with a seed Q terminates. □

As for the termination problem, several work has been proposed (e.g., [9], [1], citeKRS:88). Kifer and Lozinskii, for example, introduced a similar technique into their SYGRAF [9]. We believe that our

[3] We assume that a positive natural number k is assigned to each atom p and its call-atom cal_p as its term-depth. This assignment is arbitrary and the resulting naive evaluation with term-depth abstraction is shown to be sound and complete.

method has the same power as that of SYGRAF wrt termination, and we claim that a characteristic of AT is simplicity of its evaluation method.

6 Concluding Remarks

We have proposed a bottom-up query evaluation algorithm called Alexander Templates (AT) which is applicable to general Horn programs. We have shown that its behaviour corresponds exactly to that of the SLD-AL procedure, thus, the bottom-up computation of CP rules has the same power as the top-down evaluation of the SLD-AL procedure. Several researchers have mentioned the correspondence between the bottom-up evaluation and the top-down evaluation for some examples (e.g., [17]), but not for general Horn programs.

Although we confine ourselves to Horn programs, it is straightforward to extend AT into stratified programs [2]. We have already proposed a top-down algorithm called *OLDTNF* resolution [14] which is sound and complete for a class of *stratified* databases. Since OLDTNF resolution is OLDT resolution [15] augmented with "Negation as Failure" rule, a simple modification of AT would give a bottom-up query evaluation algorithm for stratified databases [13].

Finally, compared with previous work, the contributions of this paper can be summarized as follows :

1) A query evaluation method called the *Alexander Templates (AT)* was proposed, which is an extension of the Alexander method and is sound and complete for general Horn programs.

2) It was shown that there exists an exact one-to-one correspondence between the naive evaluation of AT and the corresponding SLD-AL procedure.

3) A technique called term-depth abstraction was introduced into the naive evaluation of CP rules, in order to guarantee its termination, whenever the naive evaluation of the original program terminates.

References

[1] F. Afrati, C. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv, and J.D. Ullman. Convergence of Sideways Query Evaluation. In *Proc. Fifth ACM Symposium on Priciples of Database Systems*, pages 24–30, 1986.

[2] K.R. Apt, H. Blair, and A. Walker. Towards A Theory of Declarative Knowledge. In J. Minker, editor, *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, pages 546–623, 1986. Washington, DC.

[3] F. Bancilhon, D. Maier, U. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. Fifth ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15, 1987.

[4] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. of the ACM-SIGMOD Conference*, pages 16–52, 1986. Washington, DC.

[5] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. Fifth ACM Symposium on Priciples of Database Systems*, pages 269–284, 1986.

[6] S.W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proc. 1987 Symposium on Logic Programming*, pages 264–272, IEEE Computer Society, 1987.

[7] T. Kanamori and T. Kawamura. *Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation*. ICOT Technical Report TR-279, ICOT, 1987.

[8] B.D. Kemp and R.W. Topor. *Completeness of a Top-down Query Evaluation Procedure for Stratified Databases*. Technical Report, Dept. of Computer Science, Univ. of Melbourne, 1988. also in 5th International Conference Symposium on Logic Programming, Seattle.

[9] M. Kifer and E. Lozinskii. Implementing Logic Programs As a Database System. In *Proc. of International Conference on Data Engineering*, pages 375–385, 1987.

[10] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[11] R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 140–159, Seattle, 1988.

[12] J. Rohmer, R. Lescouer, and J.M. Kerisit. The Alexander Method -- A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing*, 4(3):273–285, 1986.

[13] H. Seki. On the Power of Continuation Passing. manuscript, 1987. Its revised version is to appear in ICOT Technical Report.

[14] H. Seki and H. Itoh. *An Evaluation Method for Stratified Programs under the Extended CWA*. ICOT Technical Report 337, ICOT, 1988. also in 5th International Conference Symposium on Logic Programming, Seattle.

[15] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, London, 1986.

[16] L. Vieille. A Database-complete Proof Procedure Based on SLD-resolution. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 74–103, Melbourne, 1987.

[17] L Vieille. *From QSQ towards QoSaQ: Global Optimization of Recursive Queries*. Technical Report TR-KB-18, ECRC, 1987.

[18] L. Vieille. Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In *Proceedings of the First International Conference on Expert Database Systems*, pages 179–193, Charleston, 1986.

Appendix

A1: Example 2.2

We illustrate **Transformation Algorithm** described in Section 2.2, using the same generation problem in **Example 2.1**. In step 1, the following rules are introduced, each of which corresponds to rule (1) and (2), respectively.

$$sol\_sg(X,Y) \quad - \quad call\_sg(X,Y), flat(X,Y) \tag{21}$$

$$sol\_sg(X,Y) \quad - \quad call\_sg(X,Y), up(X,Z1), sg(Z1,Z2), flat(Z2,Z3), sg(Z3,Z4), down(Z4,Y). \tag{22}$$

First, consider rule (21). Since it does not satisfy the termination condition in step 2, we apply step 3 to it, generating the two rules (7) and (8) in **Example 2.2**. Similarly, we apply step 3 to (22), generating the two rules:

$$call\_up(X,Z1), cont(2\text{-}1,[X,Z1],[Y]) \leftarrow call\_sg(X,Y)$$

$$sol\_sg(X,Y) \leftarrow sol\_up(X,Z1), cont(2\text{-}1,[X,Z1],[Y]), sg(Z1,Z2), flat(Z2,Z3), sg(Z3,Z4), down(Z4,Y).$$

The first rule in the above is exactly rule (9), and it is a CP rule. Thus, it is in $CPRs(2)$. On the other hand, we apply step 2 to the second rule again, generating the following two rules:

$$call\_sg(Z1,Z2), cont(2\text{-}2,[Z1,Z2],[X,Y]) \leftarrow sol\_up(X,Z1), cont(2\text{-}1,[X,Z1],[Y])$$

$$sol\_sg(X,Y) \leftarrow sol\_sg(Z1,Z2), cont(2\text{-}2,[Z1,Z2],[X,Y]), flat(Z2,Z3), sg(Z3,Z4), down(Z4,Y).$$

The first rule gives another CP rule derived from (2). Similarly, we apply step 2 to the second rule, until the termination condition is satisfied.

A2: Comparison with AT and Supplementary MT

As mentioned in 2.2.1, we can modify our rule transformation algorithm into the one which does not transform database predicates in the body of each rule in P. The following is such an example corresponding to **Example 2.1**.

$$sol\_sg(X,Y) \quad - \quad call\_sg(X,Y), flat(X,Y) \tag{23}$$

$$call\_sg(Z1,Z2), cont(2\text{-}2,[Z1,Z2],[X,Y]) \quad - \quad call\_sg(X,Y), up(X,Z1) \tag{24}$$

$$call\_sg(Z3,Z4), cont(2\text{-}4,[Z3,Z4],[X,Y]) \quad - \quad sol\_sg(Z1,Z2), cont(2\text{-}2,[Z1,Z2],[X,Y]), flat(Z2,Z3) \tag{25}$$

$$sol\_sg(X,Y) \quad - \quad sol\_sg(Z3,Z4), cont(2\text{-}4,[Z3,Z4],[X,Y]), down(Z4,Y) \tag{26}$$

Supplementary Magic Templates (or Generalized Supplementary Magic Sets) produces the following rules by the rewriting algorithm, when it is applied to the same generation problem (we assume also the total sip here). The following rules are borrowed from [5].

$$magic\_sg(a, Y). \quad \% \text{ We assume that a query } \leftarrow sg(a, Y) \text{ is given} \tag{27}$$

$$supmagic_2^2(X, Y, Z1, Z2) \leftarrow magic\_sg(X, Y), up(X, Z1) \quad \% \text{ from rule (2)} \tag{28}$$

$$supmagic_3^2(X, Y, Z1, Z2) \leftarrow supmagic_2^2(X, Y, Z1, Z2), sg(Z1, Z2) \quad \% \text{ from rule (2)} \tag{29}$$

$$supmagic_4^2(X, Y, Z2, Z3) \leftarrow supmagic_3^2(X, Y, Z1, Z2), flat(Z2, Z3) \quad \% \text{ from rule (2)} \tag{30}$$

$$sg(X, Y) \leftarrow magic\_sg(X, Y), flat(X, Y) \quad \% \text{ Modified rule (1} \tag{31}$$

$$sg(X, Y) \leftarrow supmagic_4^2(X, Y, Z2, Z3), sg(Z3, Z4), down(Z4, Y) \quad \% \text{ Modified rule (2)} \tag{32}$$

$$magic\_sg(Z1, Z2) \leftarrow supmagic_2^2(X, Y, Z1, Z2) \quad \% \text{ from rule (2), 2nd body literal} \tag{33}$$

$$magic\_sg(Z3, Z4) \leftarrow supmagic_4^2(X, Y, Z2, Z3) \quad \% \text{ from rule (2), 4th body literal} \tag{34}$$

The seed (27) corresponds to the initial call-atom $call\_sg(a, Y)$ in AT. Each *magic_sg* (*supmagic*) in the above corresponds to *call_sg* (a cont-atom) in AT, respectively. Suppose that rule (28) generates an atom $supmagic_2^2(X, Y, Z1, Z2)$. Then, it would fire rule (33), which produces an atom $magic\_sg(Z1, Z2)$. These two rules in Supplementary MT can merged into one CP rule (24) in AT. Similarly, rules (29), (30) and (34) can merged into a CP rule (25).

A3: Example 2.3

The computation process of the naive evaluation of $CPRs(P_{sg} \cup D_{sg})$, together with initial call-atom $call\_sg(a, Y)$) is shown in Figure 1. At each iteration step, only newly generated atoms are depicted.

A4: Example 3.1

The computation process of the SLD-AL procedure for $\leftarrow sg(a, Y)$ with respect to (P_{sg}, D_{sg}) is shown in Figure 2.

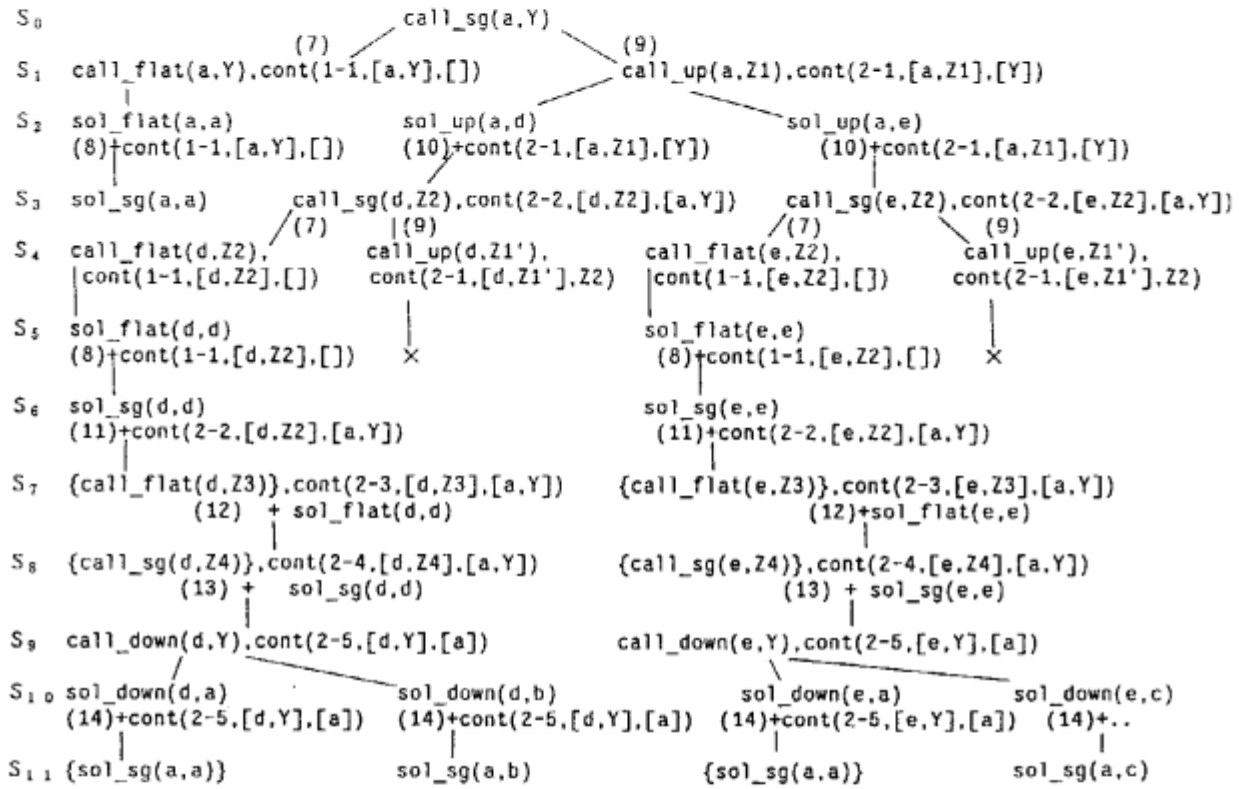


Figure 1: Naive Evaluation of $CPRs(P_{sg} \cup D_{sg})$ for seed $call\_sg(a, Y)$)

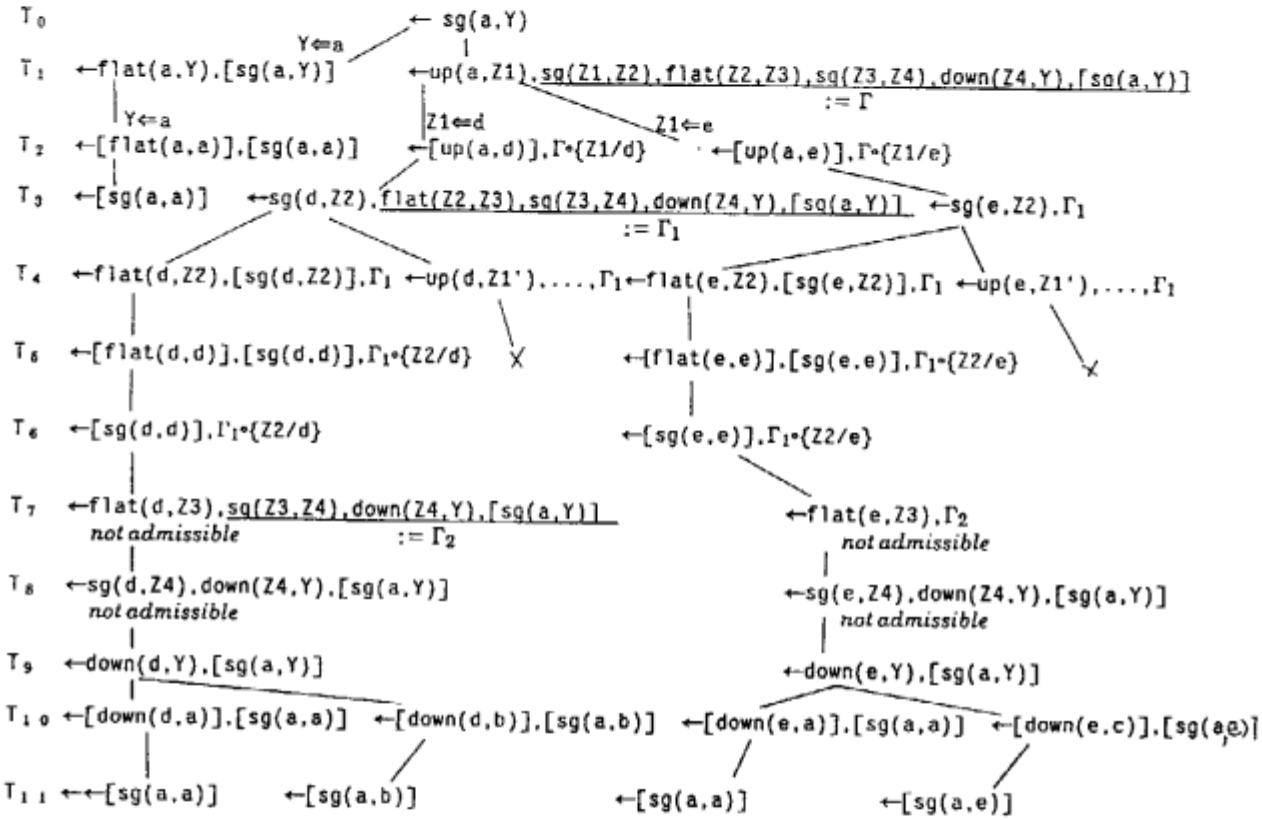


Figure 2: The SLD-AL procedure for $\leftarrow sg(a, Y)$