Deductive and Object-Oriented Databases

Proceedings of the First International Conference on
Deductive and Object-Oriented Databases (DOOD89)
Kyoto Research Park, Kyoto, Japan, 4–6 December, 1989

Edited by

Won KIM
MCC
Austin, Texas
U.S.A.

Jean-Marie NICOLAS
ECRC
Munich
Federal Republic of Germany

Shojiro NISHIO
Department of Information and Computer Sciences
Osaka University
Toyonaka, Osaka
Japan

1990

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD • TOKYO

CONTENTS

KEYNOTE ADDRESS

Towards a New Step of Logic Paradigm
K. Fuchi 3

A STATUS UPDATE ON DEDUCTIVE DATABASES

Object Identity and Inheritance in Deductive Databases:
An Evolutionary Approach (*Invited Paper*)
C. Zaniolo 7

Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled
F. Bry 25

DEDUCTIVE QUERY EVALUATION

On Termination of Datalog Programs (*Extended Abstract*)
A. Brodsky and Y. Sagiv 47

The Level-Cycle Merging Method
J. Han and L.J. Henschen 65

Distribution of Selections: The Missing Link between Strategies for
Relational Databases and Deductive Databases
N. Miyazaki 83

Combining Deduction by Certainty with the Power of Magic
H. Schmidt, N. Steger, U. Güntzer, W. Kiessling, R. Azone, and R. Bayer 103

On Deductive Query Evaluation in the DedGin* System
A. Lefebvre and L. Vieille 123

Detecting and Eliminating Redundant Derivations in Logic Knowledge Bases
A.R. Helm 145

OODB THEORY

A Theory of Functional Dependencies for Object-Oriented Data Models
G.E. Weddell 165

Object Identity, Equality and Relational Concept
Y. Masunaga 185

A Formal System for Producing Demons from Rules in an Object-Oriented Database
Y. Caseau 203

OODB FEATURES

The Object-Oriented Database System Manifesto (*Invited Paper*)
M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik 223

Meta Operations for Type Management in Object-Oriented Databases: A Lazy Mechanism for Schema Evolution
L. Tan and T. Katayama 241

A Tool Kit System for the Synthesis and the Management of Active Media Objects
Y. Tanaka 259

OODB QUERIES

Object-Oriented Queries: Equivalence and Optimization
G.M. Shaw and S.B. Zdonik 281

On Natural Joins in Object-Oriented Databases
K. Tanaka and T.-S. Chang 297

Reloop, an Algebra Based Query Language for an Object-Oriented Database System
S. Cluet, C. Delobel, C. Lécluse, and P. Richard 313

PANEL DISCUSSION

Next Generation Database Management Systems Technology
M.L. Brodie, F. Bancilhon, C. Harris, M. Kifer, Y. Masunaga, E.D. Sacerdoti, and K. Tanaka 335

DATALOG EXTENSION

Integration of Functions Defined with Rewriting Rules in Datalog
S. Grumbach 349

Possible Model Semantics for Disjunctive Databases
C. Sakama 369

The Well Founded Semantics for Disjunctive Logic Programs
K.A. Ross 385

INTEGRATING OBJECTS AND RULES

Formal Models for Object Oriented Databases (*Invited Paper*)
C. Beeri 405

HILOG: A High-Order Logic Programming Language for Non-1NF Deductive Databases
Q. Chen and W.W. Chu 431

Towards a Dedutive Object-Oriented Database Language
S. Abiteboul 453

Semantics and Evaluation of Rules over Complex Objects
A. Heuer and P. Sander 473

Inference Rules in Object Oriented Programming Systems
L. Wong 493

Features of the TEDM Object Model
D. Maier, J. Zhu, and H. Ohkawa 511

Software Process Modeling as a Strategy for KBMS Implementation
M. Jarke, M. Jeusfeld, and T. Rose 531

QUERY TRANSFORMATION

Query Optimization in Database Programming Languages
P. Valduriez and S. Danforth 553

Integrating Complex Objects and Recursion
H. Schöning 573

OOLP: A Translation Approach to Object-Oriented Logic Programming
M. Dalal and D. Gangopadhyay 593

Author Index 607

Deductive and Object-Oriented Databases
W. Kim, J.-M. Nicolas, and S. Nishio (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1990

Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled

François Bry

ECRC, Arabellastr. 17, D - 8000 Munich 81, West Germany
uucp: ...!pyramid!ecrcvax!fb

It is desirable to answer queries posed to deductive databases by computing fixpoints because such computations are directly amenable to set-oriented fact processing. However, the classical fixpoint procedures based on bottom-up reasoning – the naive and semi-naive methods – are rather primitive and often inefficient. In this article, we rely on bottom-up meta-interpretation for formalizing a new fixpoint procedure that performs a different kind of reasoning: We specify a top-down query answering method, which we call the *Backward Fixpoint Procedure.* Then, we reconsider query evaluation methods for recursive databases. First, we show that the methods based on rewriting on the one hand, and on resolution on the other hand, implement the Backward Fixpoint Procedure. Second, we interpret the rewriting of the Alexander and Magic Set methods as a specialization of the Backward Fixpoint Procedure. Finally, we argue that this rewriting is also needed for implementing efficiently the resolution-based methods. Thus, the methods based on rewriting and the methods based on resolution implement the same top-down evaluation of the original database rules by means of auxiliary rules processed bottom-up.

1. Introduction

For various reasons, fixpoint procedures are rather natural ways of processing queries posed to deductive databases. First, the declarative semantics of a set of Horn clauses can be defined as the fixpoint of an 'immediate consequence operator', as shown by van Emden and Kowalski in [vEK 76]. Moreover, although this so-called 'fixpoint semantics' is not procedural, it directly induces set-oriented query answering procedures, namely the methods that are called 'naive' and 'semi-naive' by Bancilhon and Ramakrishnan in [BAR 86]. Finally, the fixpoint theory which was developed in formal logic for studying recursive functions provides us with a useful mathematical tool for investigating query answering procedures for recursive databases.

The naive and semi-naive methods are based on rather primitive deduction techniques and are often inefficient. Indeed, both methods perform forward reasoning, i.e., they proceed bottom-up from the database rules and facts. Therefore, they do not use the constants occurring in the queries for restricting the search space. In contrast, such a restriction is a by-product of backward – or top-down – reasoning. The rewriting of the Alexander [RLK 86] and Magic Set [BMSU 86, BR 87] methods aims at achieving the same restriction on the search space with bottom-up reasoning.

In this paper, we show that it is possible to keep the advantages of processing queries through fixpoint computations, without necessarily sticking to the basic principle of the naive and semi-naive methods. We specify a new fixpoint query answering procedure, the 'Backward Fixpoint Procedure', which is based on top-down – or backward – reasoning. In other words, we apply fixpoint theory to databases with another operator than the classical immediate consequence operator of van Emden and Kowalski. The Backward Fixpoint Procedure is a sound and complete query answering method for recursive databases.

We rely on bottom-up meta-interpretation for formalizing the Backward Fixpoint Procedure, i.e., we specify a top-down evaluation of the database rules in a meta-language by means of rules intended for bottom-up processing. Meta-interpretation is a technique commonly used in Functional and Logic Programming. As we show below, bottom-up meta-interpretation permits one to obtain a surprisingly simple specification for the Backward Fixpoint Procedure.

Then, we reconsider evaluation methods for recursive databases from the viewpoint of fixpoint computation. Several methods have been proposed for evaluating queries on recursive databases. Those that ensure termination on all recursive databases defining finitely many facts – e.g., function-free databases – follow one or the other of two approaches. The methods of the first type rewrite the database rules and process the rewritten rules bottom-up. The Alexander [RLK 86] and Magic Set [BMSU 86, BR 87] methods are based on this principle. The second approach is an extension of SLD-Resolution [LLO 87] that consists of storing the encountered queries and the proven answers. The ET^* and ET_{interp} algorithms [DIE 87], OLDT-Resolution [TS 86], QSQ and SLDAL-Resolution [VIE 87], and the RQA/FQI procedure [NEJ 87], are methods of the second type. We investigate both types of methods. We show that the methods based on rewriting as well as the methods based on resolution implement the Backward Fixpoint Procedure. In other words, they express the same top-down reasoning principle in different formalisms.

Similarities between rewriting-based and resolution-based methods were already observed by many authors. In particular, Beeri and Ramakrishnan showed in [BR 87] that the same strategies – called 'sideway information passing strategies' – can be applied to optimize both types of methods. Moreover, Ramakrishnan noticed in [RAM 88] that the same propagation of constants is possible with rewriting-based and resolution-based methods. This point was investigated more formally by Ullman in [ULL 89]. Commonalities in the inferences of both types of methods were often cited – e.g., in [DR 86, BEE 89, VIE 89]. Recently, Seki established a one-to-one mapping between the inferences performed by methods of both types [SEK 89]. These observations and results are precursors of the study we present here.

Examining efficient implementations of the Backward Fixpoint Procedure, we investigate a technique called specialization. Specializing meta-interpreters is a classical way of obtaining efficient procedures from formal specifications. We show that the rewriting of the Alexander and Magic Set methods can be interpreted as a specialization of the Backward Fixpoint Procedure. We argue that this rewriting is also needed in efficient implementations of resolution-based methods. This motivates features of the implementation of SLDAL-Resolution which is reported in [LV 89]. Thus, efficient implementations of methods of both kinds have to rely on the same rewriting of the database rules and to process the rewritten rules bottom-up.

Relying on the meta-interpreter for the Backward Fixpoint Procedure, we give in [BRY 89] simple soundness and completeness proofs for the Alexander and Magic Set methods, and for the ET^* and ET_{interp} algorithms, OLDT-Resolution, QSQ and SLDAL-Resolution, and the RQA/FQI procedure. Thus, bottom-up meta-interpretation appears to be a useful formalism for theoretical investigations of query answering procedures.

The article consists of eight sections, the first of which is this introduction. In Section 2, we review background notions and introduce notations. In Section 3, we show how rules intended for bottom-up computation can be used for specifying fixpoint procedures. Then we show in Section 4 that top-down processing of queries can be performed by a fixpoint procedure. We make use of bottom-up meta-interpretation for specifying the Backward Fixpoint Procedure. In Section 5, we refine the definition of this procedure. In Section 6, we investigate implementation issues and we show that the rewritings of the Alexander and Magic Set methods are specializations of the Backward Fixpoint Procedure. Section 7 is devoted to recursive query processing methods based on SLD-Resolution. We first show that they implement the Backward Fixpoint Procedure as well. Then we show that they require the very rewriting of the Alexander and Magic Set methods. In Section 8, we summarize the results presented in the article and we indicate directions for further research.

Because of space limitations, the proofs are omitted. They can be found in the full version of this article [BRY 89].

2. Background

A deductive database is a finite set of deduction rules and facts. Given a database DB, we shall denote its subset of deduction rules by DR(DB) and its subset of facts by F(DB). Facts are ground atoms and deduction rules are expressions of the form:

$$H \leftarrow L_1 \wedge \dots \wedge L_n$$

where $n \geq 1$, H is an atom, and the L_is are literals. Such a rule denotes the formula:

$$\forall x_1 \dots \forall x_k \ (L_1 \wedge \dots \wedge L_n \Rightarrow H)$$

where the x_js are the variables occurring in H or in the L_is. If all L_is are positive literals, then the rule is called a *Horn rule*. A database is called a *Horn database* if all its rules are Horn rules. H is called the *head* of the rule. The conjunction $L_1 \wedge \dots \wedge L_n$ is called its *body*.

A dependency relationship on database predicates – or relations – is inductively defined as follows. A predicate p *depends* on each predicate occurring in the body of a rule with head predicate p, and on each predicate on which one of these body predicates depends. A predicate which depends on itself is said to be *recursive*. A database is *recursive* if one of its predicates is recursive.

Words beginning with lower case letters from the end of the alphabet (u, v, w, etc.) – with or without subscripts – denote variables. Words beginning with other lower case characters are used for denoting constants and predicates.

The Herbrand base HB(DB) of a database DB is the set of ground atoms that can be constructed from the predicate, constant, and function symbols occurring in DB. HB(DB) is finite if and only if DB contains no function symbols.

A ground atom A is said to be an *immediate consequence* of a database DB if there exist:

- a rule $H \leftarrow L_1 \wedge \dots \wedge L_n \in$ DB
- a substitution τ

such that:

- $H\tau = A$
- $L_i\tau \in$ F(DB) if L_i is a positive literal, and $L_i\tau \notin$ F(DB) otherwise.

The *immediate consequence operator* T on DB – formally, on HB(DB) $\cup$ DR(DB) – is the function associating with each D $\subseteq$ H(DB) $\cup$ DR(DB) the set T(D) of its immediate consequences.

More generally, an *operator* on a set S is a function on the power set of S. An operator Γ on a set S is *monotonic* if it satisfies the property:

$$\forall P_1 \subseteq S \quad \forall P_2 \subseteq S \qquad [\, P_1 \subseteq P_2 \;\; => \;\; \Gamma(P_1) \subseteq \Gamma(P_2) \,]$$

Restricted to Horn databases, the immediate consequence operator T is monotonic. However, T is not monotonic on non-Horn databases.

If Γ is an operator on a set S and if P $\subseteq$ S, we recall the notation:

$$\Gamma\uparrow^{\omega}(P) \;=\; \cup_{n \in \mathbf{N}} \Gamma\uparrow^{n}(P)$$

where:

$$\Gamma\uparrow^{0}(P) \;=\; P$$

$$\Gamma\uparrow^{n+1}(P) = \;\; \Gamma(\Gamma\uparrow^{n}(P)) \cup \Gamma\uparrow^{n}(P) \qquad \text{for } n \in \mathbf{N}$$

A *least fixpoint* of an operator Γ on a set S is a set $\Gamma\uparrow^{n}(S)$ ($n \in \mathbf{N}^* \cup \{\omega\}$) such that:

$$\Gamma\uparrow^{\omega}(S) \;=\; \Gamma\uparrow^{n}(S)$$

$$\Gamma\uparrow^{\omega}(S) \;\neq\; \Gamma\uparrow^{k}(S) \quad \text{for } k < n$$

A monotonic operator on a set S has a unique least fixpoint on S [TAR 55]. Therefore, T admits a unique least fixpoint on Horn databases. This fixpoint is finite if $T\uparrow^{\omega}(DB) = T\uparrow^{n}(DB)$ for some $n < \omega$. This is in particular the case if no function symbols occur in DB. The semantics of a Horn database DB is formalized by defining its true facts as the facts in the least fixpoint $T\uparrow^{\omega}(DB)$.

The least fixpoint of T on a function-free Horn database DB can be constructed by iteratively computing the sets $T\uparrow^{n}(DB)$ for increasing n. The computation halts as soon as no new facts are generated, i.e., when a step n is reached such that:

$$T(T\uparrow^{n}(DB)) \;\subseteq\; T\uparrow^{n}(DB)$$

Since the least fixpoint $T\uparrow^{\omega}(DB)$ of T on a function-free Horn database is finite, this procedure always terminates when applied to such databases. In particular, it terminates on recursive function-free Horn databases. Following Bancilhon and Ramakrishnan [BAR 86], we call this procedure the *naive method.*

A drawback of the naive method is to compute repeatedly facts that have already been generated: While computing $T\uparrow^{n+1}(DB)$, all immediates consequences of $T\uparrow^{i}(DB)$ for $0 \leq i < n$ are recomputed. Since T is monotonic on Horn databases, it suffices to generate those elements of $T\uparrow^{n+1}(DB)$ that have at least one premise in $T\uparrow^{n}(DB) \setminus T\uparrow^{n-1}(DB)$. Improving the naive method in this way results in the so-called *semi-naive method.* Various search strategies for the semi-naive method are investigated in [SKGB 87].

For the sake of simplicity, we shall consider databases without function symbols. Though simple in principle, the extension of the results presented in this article to databases with function symbols would require more sophisticated formulations of a few definitions.

3. Fixpoint Procedures as Bottom-up Meta-interpreters

In this section, we introduce the 'bottom-up meta-interpretation' technique with a quite obvious and simple example. This technique is used in more interesting ways in Section 4.

The computation of the immediate consequences T(DB) of a Horn database DB can be paraphrased as follows. For all rules $H \leftarrow A_1 \wedge ... \wedge A_n$ in DB and all substitutions τ such that $A_i\tau \in$ DB (i = 1, ..., n), the facts $H\tau$ are proved. The immediate consequence operator T can be expressed as the forward processing of the following rule:

$$\text{fact}(H) \leftarrow \text{rule}(H \leftarrow B) \wedge \text{evaluate}(B)$$

where the predicates 'rule' and 'evaluate' respectively express access to the set of deduction rules and facts. For the sake of simplicity, we assume here and in the rest of the article that bodies of rules are evaluated from left to right. Note, however, that this hypothesis is not necessary and that the results we establish do not require it.

A bottom-up evaluation of the above-defined rule produces an expression 'fact(F)' for each $F \in T(DB)$. By iterating in the naive or semi-naive manner, one generates an expression 'fact(F)' for each $F \in T\uparrow^{\omega}(DB)$. Figure 1 illustrates this principle on an example.

Database: r(a) s(a) $p(x) \leftarrow q(x) \wedge r(x)$
s(b) $q(x) \leftarrow s(x)$

Successful derivations:

Step 1: $\text{fact}(q(x)) \leftarrow \text{rule}(q(x) \leftarrow s(x)) \wedge \text{evaluate}(s(x))$ σ_1=[x:a] σ_2=[x:b]

Step 2: $\text{fact}(p(x)) \leftarrow \text{rule}(p(x) \leftarrow q(x) \wedge r(x)) \wedge \text{evaluate}(q(x) \wedge r(x))$ σ_3=[x:a]

Fig. 1

The evaluation of 'rule($H \leftarrow B$)' first binds H to 'p(x)' and B to 'q(x) $\wedge$ r(x)'. Since there are no q facts in the database, the evaluation of B fails. H and B are then respectively bound to 'q(x)' and 's(x)' from the second rule. Processing 'evaluate(B)' yields the bindings σ_1=[x:a] and σ_2=[x:b], i.e., 'fact(q(a))' and 'fact(q(b))' are proven. They are added to the database. These new facts now 'fire' the database rule '$p(x) \leftarrow q(x) \wedge r(x)$' when H is bound to 'p(x)' and B to 'q(x) $\wedge$ r(x)'. 'evaluate(B)' succeeds with the binding σ_3=[x:a]: 'fact(p(a))' is proven. The procedure stops because the most recently derived fact p(a) cannot serve as a premise in any rule.

The semantics of 'evaluate' can be formally defined as follows: If B is an atom or a conjunction of atoms and σ is a substitution of constants for variables in B, 'evaluate(B)σ' holds if and only if $B\sigma$ evaluates to true over the current facts, i.e., the database facts and the already generated 'fact' atoms. We do not specify here any procedure for 'evaluate': Let us assume that we rely on a non-deductive, relational query evaluator.

The above-defined rule is a meta-interpreter, i.e., it is a logic program that treats another logic program. namely the database under consideration, as data and interprets or runs it. Meta-programming is a common practice in Functional and Logic Programming. The vari-

ables in a meta-interpreter range over atomic and conjunctive queries. We denote them with upper case letters, in order to distinguish them from conventional variables that range over attribute values.

Let M_{DB} denote a database consisting of the above-defined deduction rule and of the two relations {rule(R) | R ∈ DR(DB)} and {fact(A) | A ∈ F(DB)}. The following proposition shows that the least fixpoint $T\uparrow^{\omega}(M_{DB})$ expresses the least fixpoint $T\uparrow^{\omega}(DB)$ of the underlying database DB.

Proposition 3.1:
Let DB be a Horn database, A a fact, and n ∈ **N***.

1. $A \in T\uparrow^{\omega}(DB)$ iff $fact(A) \in T\uparrow^{\omega}(M_{DB})$
2. $A \in T\uparrow^{n}(DB) \setminus T\uparrow^{n-1}(DB)$ iff $fact(A) \in T\uparrow^{n}(M_{DB}) \setminus T\uparrow^{n-1}(M_{DB})$

Intuitively, the second point of Proposition 3.1 means that the semi-naive computation of $T\uparrow^{\omega}(M_{DB})$ expresses the semi-naive computation of $T\uparrow^{\omega}(DB)$ in the meta-language. It follows that the above specification of the operator T by means of a rule can be viewed – and used – as an implementation, if we have at our disposal a naive or semi-naive query evaluator. This is not really interesting here, since we use the operator T itself. However, it is useful with other operators, as it permits us to run fixpoint procedures that perform deductions of other types with a semi-naive evaluator.

Specifying query answering procedures as bottom-up meta-interpreters has two main consequences, as far as the computation of fixpoints is concerned. First, terms that are not in first normal form are generated, e.g., 'fact(p(a))'. Second, non-ground terms can be generated, as happens with the Backward Fixpoint Procedure of Section 4. This requires replacing syntactical identity tests by more expensive instance tests. In Section 5, we describe a normalization technique and we show how to perform instance tests efficiently. In the next section, we shall assume that the semi-naive query evaluator at hand correctly handles unnormalized and non-ground terms.

4. The Backward Fixpoint Procedure: Principle

In the previous section, we have given a bottom-up meta-interpreter to process the object rules – i.e., the database rules – in a bottom-up manner. In this section, we show that bottom-up meta-interpretation can also be applied for specifying top-down reasoning on the object rules.

The following rules specify an operator, that we call T_b. This operator processes the database rules – accessed with the predicate 'rule' – in a top-down manner. The rule for 'fact' expresses that a body of a rule is evaluated only in case a query is posed on the head of that rule. The top-down evaluation principle is rather clearly recognizable in the rules for '$query_b$': The first $query_b$-rule for example induces a query on the body of a rule from a query on its head. The last two rules split conjunctive queries into atomic ones in order to permit the top-down expansion of these atomic expressions with the first $query_b$-rule.

(i)	$fact(Q)$	←	$query_b(Q) \wedge rule(Q \leftarrow B) \wedge evaluate(B)$
(ii)	$query_b(B)$	←	$query_b(Q) \wedge rule(Q \leftarrow B)$
(iii)	$query_b(Q_1)$	←	$query_b(Q_1 \wedge Q_2)$
(iv)	$query_b(Q_2)$	←	$query_b(Q_1 \wedge Q_2) \wedge evaluate(Q_1)$

The predicate 'evaluate' expresses access to the already generated facts, as in the rule for the immediate consequence operator T given in Section 3.

We call 'Backward Fixpoint Procedure' the procedure that, applied to a Horn database DB and to a set Q of $query_b$-atoms, computes the least fixpoint $T_b \uparrow^{\omega}(DB \cup Q)$ of the operator T_b on DB and Q. The atoms in Q are the initial queries posed to the database DB. Figure 2 shows on an example how the Backward Fixpoint Procedure computes $T_b(DB \cup Q)$. Note that no t-facts are derived.

<u>Database:</u>

r(a)	s(a)	u(a)	$p(x) \leftarrow q(x) \wedge r(x)$
	s(b)	u(b)	$q(x) \leftarrow s(x)$
			$t(x) \leftarrow s(x) \wedge u(x)$

<u>Queries:</u> $query_b(p(b))$ $query_b(q(x))$

<u>Successful derivations:</u>

Step 1:

fact(q(x))	$\leftarrow$	$query_b(q(x)) \wedge rule(q(x) \leftarrow s(x)) \wedge evaluate(s(x))$	σ_1=[x:a] σ_2=[x:b]
$query_b(q(x) \wedge r(x))$	$\leftarrow$	$query_b(p(b)) \wedge rule(p(x) \leftarrow q(x) \wedge r(x))$	σ_3=[x:b]
$query_b(s(x))$	$\leftarrow$	$query_b(q(x)) \wedge rule(q(x) \leftarrow s(x))$	σ_4=[]

Fig. 2

Evaluating the body of rule (i) first binds '$query_b(Q)$' to '$query_b(p(b))$', B to '$q(x) \wedge r(x)$' and yields the binding [x:b]. B is not satisfied by the database facts: No facts are generated. '$query_b(Q)$' from rule (i) is then bound to '$query_b(q(x))$', and B to 's(x)'. The evaluation of B over the database facts yields the bindings σ_1=[x:a] and σ_2=[x:b], thus generating 'fact(q(a))' and 'fact(q(b))'. Rule (ii) generates from '$query_b(p(b))$' the expression '$query_b(q(x) \wedge r(x))$' with the binding σ_3=[x:b]. Similarly, '$query_b(s(x))$' is derived by rule (ii) from '$query_b(q(x))$'.

It is reasonable to evaluate the bodies of rules (i)-(iv) from left to right. With this ordering, the $query_b$-atoms constrain the evaluations. In rule (iv), this ordering ensures that Q_1 is bound to an atom when 'evaluate(Q_1)' is processed. With another ordering, the type of the variable Q_1, i.e., the set of database queries, would have to be searched. Evaluating the conjunction

$$rule(Q \leftarrow B) \wedge evaluate(B)$$

before '$query_b(Q)$' in rule (i) would be inefficient because useless 'evaluate(B)' expressions would be processed. However, this inefficient ordering would not compromise the top-down paradigm: The useless values would be filtered out during the evaluation of '$query_b(Q)$'.

It is worth noting that, although based on backward reasoning like Linear and SLD-Resolution [LLO 87], the Backward Fixpoint Procedure differs significantly from these methods. A fundamental difference with SLD-Resolution is that new answers

generated with the Backward Fixpoint Procedure – i.e., new values for the relation 'fact' – may trigger the generation of new queries – i.e., new values for the relation '$query_b$'.

For example, an expression

$$query_b(p(x) \wedge q(x, y))$$

can be generated during the computation of $T\uparrow^n(DB)$ at a time where p-facts have not yet been generated. The generation of a fact 'p(a)' at step m > n induces from the previously computed $query_b$-expression a term '$query_b(q(a, y))$' during the computation of $T\uparrow^{m+1}(DB)$. In contrast, SLD-Resolution would have to recompute the expression

$$query_b(p(x) \wedge q(x, y))$$

in order to generate '$query_b(q(a, y))$' once 'p(a)' is obtained. In order to ensure termination on recursive databases, the query answering procedures based on SLD-Resolution collect queries and answers, in the same way as the Backward Fixpoint Procedure does.

The following proposition establishes the soundness and completeness of the Backward Fixpoint Procedure.

Proposition 4.1:
Let DB be a Horn database, A an atom, and τ a substitution such that $A\tau$ is ground.

$A\tau \in T\uparrow^{\omega}(DB)$ iff $fact(A\tau) \in T_b\uparrow^{\omega}(DB \cup \{query_b(A)\})$

The semi-naive method always terminates on databases defining finitely many facts, even if they are recursive. Therefore, so does the Backward Fixpoint Procedure. It follows from Proposition 4.1 that:

Corollary 4.1:
The Backward Fixpoint Procedure is a sound and complete query answering method for (possibly recursive) Horn databases.

It is a terminating query answering method for databases defining finitely many facts – e.g., function-free databases.

5. The Backward Fixpoint Procedure Revisited

A direct implementation of rules (i)-(iv) can induce undesirable redundancies. Consider for example a database containing a rule '$p \leftarrow q \wedge r$' and the query 'p'. The following instances of the rules (i)-(iv) are relevant:

$fact(p)$	$\leftarrow$	$query_b(p) \wedge rule(p \leftarrow q \wedge r) \wedge evaluate(q \wedge r)$	*from*	(i)
$query_b(q \wedge r)$	$\leftarrow$	$query_b(p) \wedge rule(p \leftarrow q \wedge r)$	*from*	(ii)
$query_b(q)$	$\leftarrow$	$query_b(q \wedge r)$	*from*	(iii)
$query_b(r)$	$\leftarrow$	$query_b(q \wedge r) \wedge evaluate(q)$	*from*	(iv)

Both the first and the last rules consult the facts for 'q'. This access can be shared by refining the specification of the predicate 'evaluate'.

We replace the unary predicate 'evaluate' by a binary one, whose arguments respectively denote the already evaluated part of a conjunctive query, and the rest of the query. Thus, an expression 'evaluate(∅, Q)' denotes a completely non-evaluated query 'Q'. By contrast, 'evaluate(B)' in rules (i) ('evaluate(Q_1)' in rule (iv), resp.) must be replaced by 'evaluate(B, ∅)' ('evaluate(Q_1, ∅)', resp.) which denotes a completed evaluation of B (Q_1, resp.). The following bottom-up rules specify the binary predicate 'evaluate':

(v)	evaluate(∅, B)	←	$\text{query}_b(Q) \wedge \text{rule}(Q \leftarrow B)$
(vi)	evaluate(B_1, B_2)	←	$\text{evaluate}(\emptyset, B_1 \wedge B_2) \wedge \text{fact}(B_1)$
(vii)	evaluate($B_1 \wedge B_2$, B_3)	←	$\text{evaluate}(B_1, B_2 \wedge B_3) \wedge B_1 \neq \emptyset \wedge \text{fact}(B_2)$
(viii)	evaluate(B, ∅)	←	fact(B)
(ix)	evaluate($B_1 \wedge B_2$, ∅)	←	$\text{evaluate}(B_1, B_2) \wedge B_1 \neq \emptyset \wedge \text{fact}(B_2)$

Let T'_b be the operator specified by the rules (i)-(ix) – an expression 'evaluate(X)' being replaced in rule (i) and (iv) by 'evaluate(X, ∅)'.

Proposition 5.1:
Consider a database DB and a set of query_b-facts Q. Let B_1 denote either ∅, or an atom, or a conjunction of atoms. Let B_2 and B_3 denote atoms or conjunctions of atoms. Let $n \in \mathbf{N}$.

$$\exists B_1 \ \exists B_3 \quad \text{evaluate}(B_1, B_2 \wedge B_3) \in T'_b\uparrow^{\omega}(DB \cup Q) \quad \text{iff} \quad \text{query}_b(B_2) \in T'_b\uparrow^{\omega}(DB \cup Q)$$

By Proposition 5.1, rules (ii)-(iv) can be replaced by the following rules, without affecting the semantics of the operator T'_b.

(x)	$\text{query}_b(B_2)$	←	$\text{evaluate}(B_1, B_2) \wedge B_2 \neq (C_1 \wedge C_2)$
(xi)	$\text{query}_b(B_2)$	←	$\text{evaluate}(B_1, B_2 \wedge B_3)$

Finally, we prove the equivalence of the operator T'_b specified by rules (i) and (v)-(xi) and the operator T_b specified by rules (i)-(iv).

Proposition 5.2:
Let DB be a Horn database, Q a set of query_b-facts, A an atom, and τ a substitution such that Aτ is ground.

$$\text{evaluate}(A\tau, \emptyset) \in T'_b\uparrow^{\omega}(DB \cup Q) \quad \text{iff} \quad \text{fact}(A\tau) \in T_b\uparrow^{\omega}(DB \cup Q)$$

When it is not otherwise stated, we shall not distinguish any more between T_b and T'_b, and we shall implicitly refer to the last specification of the Backward Fixpoint Procedure, i.e., the specification by means of rules (i) and (v)-(xi).

6. Specialization: The Logic of Magic

Two difficulties are encountered when implementing the Backward Fixpoint Procedure. The meta-interpreter which specifies it, on the one hand relies on structures like 'query_b(p(a, b))' that are not in first normal form, i.e., that contain nested terms. On the other hand, it generates non-ground tuples such as 'query_b(p(x, b))'.

First, we show that normalized structures can be obtained by relying on a technique called 'specialization'. We consider an encoding of variables by means of ground expressions and we show that a specialization also permits us to perform this encoding at compile time. Then, we apply these specializations to the Backward Fixpoint Procedure. This yields the rewriting algorithms of the Alexander and Magic Set methods.

6.1. Normalization by Specialization

Consider rule (i) of the Backward Fixpoint Procedure:

$$\text{(i)}\quad \text{fact(Q)} \leftarrow \text{query}_b\text{(Q)} \wedge \text{rule(Q} \leftarrow \text{B)} \wedge \text{evaluate(B)}$$

It can be specialized with respect to a database DB by pre-evaluating the expression 'rule(Q ← B)' over the rules in DB. Doing so, each rule in DB yields one partially instantiated version of (i). For example, a database rule 'p(x) ← q(x) ∧ r(x)' yields:

$$\text{fact(p(x))} \leftarrow \text{query}_b\text{(p(x))} \wedge \text{evaluate(q(x)} \wedge \text{r(x))}$$

which can be simplified into:

$$\text{p(x)} \leftarrow \text{query}_b\text{(p(x))} \wedge \text{q(x)} \wedge \text{r(x)}$$

The expression 'query_b(p(x))' can similarly be normalized by specializing the predicate 'query_b' with respect to the relation 'p' into a predicate 'query_b-p':

$$\text{p(x)} \leftarrow \text{query}_b\text{-p(x)} \wedge \text{q(x)} \wedge \text{r(x)}$$

Such a normalization by means of rule and predicate specialization is a kind of 'partial evaluation'. Partial evaluation techniques are commonly applied in artificial intelligence [SES 87].

By the following lemma, normalization by specialization does not affect the semantics of a database. Given a database DB, let BFP_{DB} denote the set of rules obtained by evaluating the 'rule' expressions in the rules (i)-(iv) that specify the Backward Fixpoint Procedure over the database rules in R(DB). Given a set Q of query_b-atoms, let $N(\text{BFP}_{DB})$ denote the set of rules and facts obtained from $R_{DB} \cup Q$ by applying the following rewriting rules, where p denotes a database predicate and $\vec{x}$ a list of terms:

fact(F)	→	F
evaluate(B)	→	B
$\text{query}_b(\text{p}(\vec{x}))$	→	$\text{query}_b\text{-p}(\vec{x})$

<u>Lemma 6.1:</u>
Let DB be a database, Q a set of query_b-atoms, and $n \in \mathbf{N} \cup \{\omega\}$.

1. $\text{fact}(\text{p}(\vec{x})) \in T_b\!\uparrow^n\!(\text{DB})$ iff $\text{p}(\vec{x}) \in T_b\!\uparrow^n\!(N(\text{BFP}_{DB}))$

2. $\text{query}_b(\text{p}(\vec{x})) \in T_b\!\uparrow^n\!(\text{DB})$ iff $\text{query}_b\text{-p}(\vec{x}) \in T_b\!\uparrow^n\!(N(\text{BFP}_{DB}))$

The improved version of the Backward Fixpoint Procedure given in Section 5, i.e., the specification by means of rules (i) and (v)-(xi), relies on a binary predicate 'evaluate'. The normalization by specialization of rules (i) and (v)-(xi) therefore requires a more sophisticated rewriting than the one given above. This rewriting is introduced below, in Section 6.3. Lemma 6.1 also holds for this refined rewriting.

6.2. Pre-encoding of Variables

The Backward Fixpoint Procedure may generate non-ground tuples. Non-ground tuples are undesirable for two reasons. On the one hand, the elimination of logical duplicates has to rely on full unification instead of syntactical identity. Indeed, although they are syntactically different, the non-ground tuples '$query_b(p(x))$' and '$query_b(p(y))$' are logically equivalent. On the other hand, non-ground tuples either have to be encoded, or special file systems are needed for storing non-encoded tuples.

Non-ground expressions can be represented in terms of ground expressions by encoding the variables with ground values. One way of doing this is to reserve special symbols, not available in the user language, for this usage. Thus, a non-ground tuple 'p(x, y, a)' is rewritten into the ground tuple 'p(*, *, a)', assuming that '*' denotes the reserved constant used for encoding variables.

Such an encoding is not completely faithful, for distinct tuples like 'p(x, y, a)' and 'p(z, z, a)' are represented identically. In order to faithfully encode the constellation of variables, different codes – e.g., *1, *2, etc. – for different variables are needed. This permits for example to encode the tuple 'p(x, y, a)' as 'p(*1, *2, a)', the tuple 'p(z, z, a)' as 'p(*1, *1, a)'.

The following proposition shows that it is possible to rely on matching – or half-unification – for checking if a non-ground expression is subsumed by an expressions the variables of which are faithfully encoded.

> Proposition 6.1:
> Let DB be a database, A and B non-ground atoms, and B_c a faithful encoding of B – i.e., an instance $B\sigma$ of B such that the substitution σ uniquely assigns to each variable in B a constant '*i' which is not in the language of DB.
>
> B subsumes A if and only if A and B_c match.

Proposition 6.1 shows that encoding variables is useful not only for storing non-ground tuples with conventional file systems, but also for performing subsumption tests efficiently. Examples better treated with faithful encoding are discussed in [ULL 89].

However, it is a debatable question, whether or not the overhead of faithful encoding pays off. We do not discuss this issue here, and we assume in the sequel that an encoding with a single reserved symbol suffices.

Using the notation introduced by Ullman in [ULL 85], an encoded term 'p(*, *, a, *, c)' is written '$p^{ffbfb}(a, c)$', where the adornment 'ffbfb' expresses that the first two attributes are variables ('f' stands for *free*), the third is the constant 'a' ('b' stands for *bound*), etc. Expressed either with reserved symbols or with adornments, the encoding of variables can be pre-computed by specializing the rules. Assume that predicates with subscript 'v' may have non-ground facts. Consider the following rule:

$$p_v(x, z) \leftarrow q(x, y) \wedge r_v(y, z)$$

If z is bound during the evaluation of '$r_v(y, z)$', then it is bound in $p_v(x, z)$. Otherwise, it is free. The relation 'r_v' can be specialized into four relations 'r_v^{bb}', 'r_v^{bf}', 'r_v^{fb}', and 'r_v^{ff}' denoting respectively the various possible patterns of free variables in an r_v-tuple. The specialization of 'r_v' induces among others the following specialized rule for 'p_v':

$$p_v^{bb}(x, z) \leftarrow q(x, y) \wedge r_v^{bb}(y, z)$$

Such a transformation of rules performs the encoding of variables once, during rule specialization. It is far less efficient to perform it each time a non-ground tuple is generated. The specialization of rules according to the patterns of instantiated variables can serve other purposes than the encoding of variables.

It is in general also used for enforcing an optimal propagation of constants during the evaluation of bodies of rules, by reordering the body literals. This can be viewed as a compilation ahead of time of 'selection functions'. By Corollary 4.1 this optimization is not necessary for the correctness, the completeness, or the termination of the method.

6.3. Specialization of the Backward Fixpoint Procedure

Consider a Horn database DB. We assume that the rules in DB are assigned unique identifiers (1), (2), etc. Consider a rule labeled (k) in DB. The general form of a database rule is:

$$(k) \quad p(\vec{x}_0) \leftarrow q_1(\vec{x}_1) \wedge ... \wedge q_j(\vec{x}_j) \wedge ... \wedge q_1(\vec{x}_n)$$

where $n \in N^*$, and where the $\vec{x}_i$s denote lists of terms. Let us denote the body of this rule by:

$$\wedge_{m=1}^{m=n} q_m(\vec{x}_m)$$

The specialization with respect to (k) of the rules specifying the Backward Fixpoint Procedure refers to the the body of rule (k) and to beginning subparts of it:

$$\wedge_{m=1}^{m=j} q_m(\vec{x}_m) \quad (1 \leq j \leq n)$$

We shall denote such a begining subpart by the pair (k, j). This characterization is not ambiguous since, by hypothesis, the database rules are assigned unique identifiers.

Proposition 6.2:
Specializing the rules (i) and (v)-(xi) of the Backward Fixpoint Procedure with respect to a database rule (k) $p(\vec{x}_0) \leftarrow \wedge_{m=1}^{m=n} q_m(\vec{x}_m)$ yields the following rules:

(a^k) $p(\vec{x}_0) \leftarrow query_b\text{-}p(\vec{x}_0) \wedge evaluate(k, n, \vec{x})$ *from* (i)
(b^k) $evaluate(k, 0, \vec{x}) \leftarrow query_b\text{-}p(\vec{x}_0)$ *from* (v)

For j = 0, ..., n - 1:
(c_j^k) $evaluate(k, j+1, \vec{x}) \leftarrow evaluate(k, j, \vec{x}) \wedge q_{j+1}(\vec{x}_{j+1})$ *from* (vi)-(ix)
(d_j^k) $query_b\text{-}q_{j+1}(\vec{x}_{j+1}) \leftarrow evaluate(k, j, \vec{x})$ *from* (x)-(xi)

Figure 3 (on next page) illustrates the specialization of the Backward Fixpoint Procedure on an example. As usual, the base relations 'r' and 's' are not specialized with adornments.

The direct generation of the adorned form of the rules of Proposition 6.2 from a database rule is precisely the rewriting procedure of the Alexander method and of the Supplementary Magic Set method, the improved version of the Magic Set method given in [BR 87]. In other words, the Alexander and the Supplementary Magic Set methods implement the Backward Fixpoint Procedure by specializing its meta-interpretative specification with respect to the database rules. Like the Backward Fixpoint Procedure, these methods perform top-down processing of the original, non-rewritten database rules.

By Lemma 6.1 and Proposition 6.2, it follows from Proposition 4.1 that:

Corollary 6.1
The Alexander, Magic Set, and Supplementary Magic Set methods are sound and complete query answering methods for (possibly recursive) Horn databases.

They are terminating methods for databases defining finitely many facts – e.g., function-free databases.

The representation of begining parts of rule bodies by a pair (k, j) in the specialized rules of Proposition 6.2 is a simple means for normalizing expressions containing conjunctions.

Omitting the lists of terms $\vec{x}$ or $\vec{x}_j$ in the 'evaluates' terms would compromise the propagation of constants during the evaluation of bodies of rules. The Alexander method does keep the lists $\vec{x}$, while the Magic Set method does not. The Supplementary Magic Set method has been proposed for remedying this deficiency. In fact, the Supplementary Magic Set method re-expresses the Alexander method in a different terminology. A '$query_b$-p' predicate is called 'problem-p' in the Alexander method, while it is called 'magic-p' in the Magic Set method. The 'evaluate' atoms correspond to the 'continuations' of the Alexander method and to the 'supplementary-magic' atoms of the Supplementary Magic Set method.

Database rules:
(1) $p^b(x) \leftarrow q^b(x) \wedge r(x)$
(2) $q^b(x) \leftarrow s(x)$.

Specialization of the Backward Fixpoint Procedure:

$p^b(x) \leftarrow query_b\text{-}p^b(x) \wedge evaluate(1,2,x)$	$q^b(x) \leftarrow query_b\text{-}q^b(x) \wedge evaluate(2,1,x)$
$evaluate(1,0,x) \leftarrow query_b\text{-}p^b(x)$	$evaluate(2,0,x) \leftarrow query_b\text{-}q^b(x)$
$evaluate(1,1,x) \leftarrow evaluate(1,0,x) \wedge q^b(x)$	$evaluate(2,1,x) \leftarrow evaluate(2,0,x) \wedge s(x)$
$evaluate(1,2,x) \leftarrow evaluate(1,1,x) \wedge r(x)$	
$query_b\text{-}q^b(x) \leftarrow evaluate(1,0,x)$	$query_b\text{-}s^b(x) \leftarrow evaluate(2,0,x)$

Fig. 3

---

7. From SLD-Resolution to Fixpoint Computation

In this section, we consider the algorithms *ET** and ET_{interp} [DIE 87], OLDT-Resolution [TS 86], QSQ or SLDAL-Resolution [VIE 87], and the procedure RQA/FQI [NEJ 87]. All these methods are based on SLD-Resolution [LLO 87] and extend it in the same way. We first investigate the differences between SLD-Resolution and the Backward Fixpoint Procedure. Then, we show that the above-mentioned procedures basically remove these differences. Finally, we argue that efficient implementations of resolution-based methods must rely on the rewriting of Proposition 6.2 and process the rewritten rules bottom-up.

Applied to Horn databases, SLD-Resolution evaluates an atomic query Q by trying to unify it with database facts or heads of rules. A unification with a fact yields an immediate answer. A unification with the head of a rule in turns entails the evaluation of the rule body. Conjunctive bodies are evaluated atom after atom, following the ordering specified by a 'selection function', e.g., strictly left to right.

This approach is very similar to the Backward Fixpoint Procedure. In order to evaluate the same query, SLD-Resolution and the Backward Fixpoint Procedure in fact access the same database rules and pose the same queries. Therefore, the rules that specify the Backward Fixpoint Procedure can be viewed as a logical specification of SLD-Resolution, in the case of Horn databases.

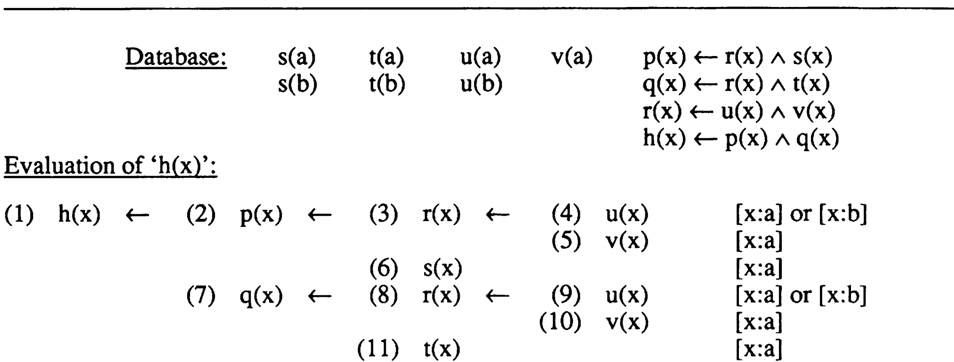

Fig. 4

However, although SLD-Resolution and the Backward Fixpoint Procedure are based on the same 'logic', they do not apply the same 'control', in the sense of Kowalski's well-known equation: Algorithm = Logic + Control. In contrast to the Backward Fixpoint Procedure, SLD-Resolution does not share results between different evaluations. Consider the example of Figure 4. In order to answer the query 'h(x)', the Backward Fixpoint Procedure shares the evaluation of the query 'r(x)' between the processing of 'p(x)' and 'q(x)'. It does not expand the proof trees rooted at 'r(x)' twice. SLD-Resolution expands it first at node (3), and re-expands it at node (8).

The difference between both procedures can be explained in terms of data structures. The Backward Fixpoint Procedure collects generated queries and proven facts in relations. Therefore, identical queries occurring in distinct parts of a proof tree are merged (this merging is the 'admissibility test' of resolution-based methods). By contrast, SLD-Resolution relies on a hierarchical data structure that relates proven facts and generated queries to the queries they come from.

In order to make clear the commonalities as well as the differences between the Backward Fixpoint Procedure and SLD-Resolution, we specify the latter method in the formalism of bottom-up meta-interpretation. We express the hierarchical data structure by labeling generated queries and proven facts.

Although a faithful expression of SLD-Resolution should be based on the version of the Backward Fixpoint Procedure given in Section 5 (rules (i) and (v)-(xi)), we consider the version of Section 4 (rules (i)-(iv)) for the sake of simplicity:

$$\text{fact}_r(Q, L) \leftarrow \text{query}_r(Q, I, L) \wedge \text{rule}(Q \leftarrow B) \wedge \text{evaluate}_r(B, I) \quad \textit{from (i)}$$

$$\text{query}_r(Q_1 \wedge Q_2, [I \mid L]) \leftarrow \text{query}_r(Q, I, L) \wedge \text{rule}(Q \leftarrow Q_1 \wedge Q_2) \quad \textit{from (ii)}$$

$$\text{query}_r(Q_1, J, [I \mid L]) \leftarrow \text{query}_r(Q, I, L) \wedge \text{rule}(Q \leftarrow Q_1) \wedge Q_1 \neq (C_1 \wedge C_2) \wedge \text{new-identifier}(J) \quad \textit{from (ii)}$$

$$\text{query}_r(Q_1, J, L) \leftarrow \text{query}_r(Q_1 \wedge Q_2, L) \wedge \text{new-identifier}(J) \quad \textit{from (iii)}$$

$$\text{query}_r(Q_2, J, L) \leftarrow \text{query}_r(Q_1 \wedge Q_2, L) \wedge \text{fact}_r(Q_1, L) \wedge \text{new-identifier}(J) \quad \textit{from (iv)}$$

An expression '$\text{fact}_r(F, L)$' relates a proven fact F to the queries it contributes to answer: The list L consists of the identifiers of these queries. For example

$$\text{fact}_r(r(a), [3, 2, 1])$$

denotes the first evaluation of 'r(x)' in the example of Figure 4. The ternary predicate 'query_r' associates with a query Q its identifier and the identifiers of the of queries it comes from. Thus the two 'r(x)' queries in Figure 4 are respectively represented by:

$$\text{query}_r(r(x), 3, [2, 1])$$

$$\text{query}_r(r(x), 8, [7, 1])$$

Conjunctive queries are similarly related to the atomic queries they come from (no identifiers are given to conjunctive queries). [I | L] denotes the list obtained by adding the identifier I in front of the list L. An initial query Q is expressed as:

$$\text{query}_r(Q, 1, [])$$

The 'evaluate_r' predicate is defined as follows: If B is an atom or a conjunction of atoms and σ is a substitution, '$\text{evaluate}_r(B, I)\sigma$' holds if and only if $B\sigma$ evaluates to true over the facts that are labeled by I or that are explicit in the database.

The 'new-identifier' expression is a call to a procedural subroutine which returns a new identifier.

In an actual implementation of SLD-Resolution, the dependencies between queries are implicitly expressed by the data structure. If a depth-first strategy is chosen, a stack suffices to express it. PROLOG interpreters, for example, rely on this data structure. In the example of Figure 4, the stack would be successively [1], [2, 1], [3, 2, 1], [4, 3, 2, 1], [5, 3, 2, 1], [6, 2, 1], etc.

As opposed to the Backward Fixpoint Procedure, SLD-Resolution is incomplete for querying recursive databases: The extension that was proposed in [DIE 87, TS 86, VIE 87, NEJ 87] achieves completeness by preventing reprocessing of queries that were already answered, and by evaluating these queries over the facts that were proven. In terms of the above-defined rules, this extension consists on the one hand of tracking the generated query_r-atoms that coincide on the first argument, and on the other hand of modifying the definition of 'evaluate_r' so that the identifiers are no longer considered. Clearly, this extension can be specified by simply removing the query identifiers and query dependency lists, i.e., by the Backward Fixpoint Procedure.

Since the resolution-based methods can be specified by the Backward Fixpoint Procedure, we have from Proposition 4.1:

Corollary 7.1:
The algorithms ET^* and ET_{interp}, OLDT-Resolution, QSQ, SLDAL-Resolution, and the RQA/FQI procedure are sound and complete query answering methods for (possibly recursive) Horn databases.

They are terminating query answering methods for databases defining finitely many facts – e.g., function-free databases.

The fixpoint formalism is useful to understand the differences between some resolution-based methods. In this formalism, the resolution-based methods are viewed as computing a fixpoint on answers *and* queries. In [DIE 87] an incomplete algorithm, called *ET*, is considered for defining the complete methods ET^* and ET_{interp}. The algorithm *ET* corresponds to the procedure QSQ as it is defined in [VIE 86] – QSQ is corrected in [NEJ 87] and [VIE 87]. The reason for incompleteness is that queries are generated only during the first round. During the subsequent rounds, the fixpoint is performed on answers only. Completeness requires treating answers and queries similarly, i.e., computing a fixpoint on *both* answers and queries.

Also, the difference between the so-called recursive and iterative versions of QSQ [VIE 86] lies in different processing of queries and answers: Recursive QSQ applies the semi-naive optimization to both, queries and answers, while Iterative QSQ applies it only to queries and does not eliminate answers that are not new. Clearly, the former approach is more efficient than the latter. This was experimentally observed in [BAR 86]. Like completeness, efficiency requires treating answers and queries similarly.

The formalization of resolution-based as well as rewriting-based methods in terms of the same procedure yields the following questions. In order to achieve an efficient implementation of one of these methods is it desirable to:

1. structure hierarchically the encountered queries following their generation?

2. rely on a semi-naive query evaluator?

3. rely on the rewriting of the Alexander or Supplementary Magic Set method?

We think that the first question must be answered negatively, the other two positively, for the following reasons:

1. A hierarchical data structure that follows the way in which the queries are generated could make their retrieval more complicated. In particular, such a structure would induce an overhead for the so-called 'admissibility test', i.e., for checking if an encountered query is new.

Moreover, a great advantage of relying on a relational data structure is to build on other components of the database management system. This makes it easier to store large sets of queries on secondary memory. Also, this permits centralized control of main memory resources.

2. It is not mandatory to rely on a language of bottom-up rules for implementing a fixpoint procedure. However, the optimization principle that distinguishes the semi-naive from the naive method is needed for the sake of efficiency. As discussed in Section 3, fixpoint procedures can be formalized in terms of bottom-up rules in a rather natural manner. No gains in efficiency seem to be reachable by changing the rule syntax on which a semi-naive procedure relies to some other, e.g., the equational syntax which is conventionally used in mathematics.

Moreover, relying on a semi-naive evaluator has the advantage of using a component of the system that is useful for efficiently processing queries that do not give rise to constant propagation, e.g., for materializing the whole of a relation. The various search strategies of the resolution-based methods – depth-first, breadth-first, and their multi-stage versions – are as well obtainable with a semi-naive method. They are investigated in [SKGB 87].

Finally, relying explicitly on a semi-naive query evaluator allows us to process some rules top-down, others bottom-up, during the same query evaluation process: It suffices not to rewrite the rules whose bottom-up evaluation is desired. This is a very simple way to implement sophisticated query optimization strategies.

3. The rewriting of the Alexander and Supplementary Magic Set methods results from the specialization of the Backward Fixpoint Procedure with respect to the database rules, as shown in Section 5. There, we justified it by showing that it permits on the one hand to normalize nested terms, and on the other hand to pre-encode the variables occurring in the generated queries. The rationale of normalization is to simplify the data structures and to permit one to rely on well-established file systems.

As we have observed, it is more efficient to pre-encode variables than to do it repeatedly when query-tuples are generated. Pre-encoding is possible only if auxiliary predicates – the '$query_b$' predicate of the Backward Fixpoint Procedure – are introduced. Indeed, these auxiliary predicates give rise to distinguishing queries that are amenable to encoding from the atoms that must be kept unchanged in order to permit their later evaluation. This justifies the introduction of the '$query_b$' expressions – i.e., the 'problem' atoms of the Alexander method or the 'magic' atoms of the Magic Set method.

The remaining feature of the rewriting, the ternary predicate 'evaluate' of Proposition 5.2 – i.e., the 'continuation' or 'supplementary magic' atoms – is justified by efficiency considerations, as discussed in [RLK 86], in [BR 87], and more briefly in Section 6.

An additional advantage of the rewriting of the Alexander and Magic Set methods is not to have to distinguish between tuples that express answers and tuples that express queries. This simplifies the procedure as well as the data structure.

8. Conclusion

During the last five years, several methods have been proposed for evaluating queries on recursive databases. Those that ensure termination on recursive databases defining finitely many facts follow one or the other of two approaches. The methods of the first type rewrite the database rules and process the rewritten rules bottom-up. This is how the Alexander [RLK 86] and Magic Set [BMSU 86, BR 87] methods proceed. The second approach is an extension of SLD-Resolution that consists of storing the encountered queries and the proven answers. It has been proposed in [DIE 87] with the ET^* and ET_{interp} algorithms, in [TS 86] with OLDT-Resolution, in [VIE 87] with QSQ and SLDAL-Resolution, and in [NEJ 87] with the RQA/FQI procedure.

On the one hand, the bottom-up processing of the first approach is often opposed to the top-down reasoning principle of the second. On the other hand, strong similarities between the two approaches were often observed. However, Beeri and Ramakrishnan noted:

> "So far there is no uniform framework in terms of which these strategies may be described and compared, and the basic ideas that are common to these strategies remain unclear"

in an article [BR 87] giving, with the notion of 'sideway information passing strategy', a first contribution towards such a framework.

In this article, we have proposed a common framework. We relied on the concept of fixpoint procedure for comparing the rewriting-based and the resolution-based methods. We showed that fixpoint theory can be applied to databases with other operators than the bottom-up reasoning immediate consequence operator of van Emden and Kowalski [vEK 76].

We specified a fixpoint query answering procedure, which we call the *Backward Fixpoint Procedure*. This procedure performs top-down reasoning but it is specified by a bottom-up meta-interpreter, i.e., in a meta-language by means of rules intended for bottom-up processing. The Backward Fixpoint Procedure was shown to be a sound and complete query answering method for recursive databases.

Then, we interpreted the Alexander and Magic Set methods on the one hand, the algorithms ET^* and ET_{interp}, OLDT-Resolution, QSQ, SLDAL-Resolution, and the procedure RQA/FQI on the other hand, in terms of the Backward Fixpoint Procedure. We showed that all these methods implement the Backward Fixpoint Procedure. Roughly speaking, rewriting-based and resolution-based methods are no longer distinguishable when expressed as fixpoint procedures in the formalism of meta-interpretation.

More precisely, we first showed that the rewriting of the Alexander and Magic Set method results from specializing the Backward Fixpoint Procedure with respect to the database rules. Then, investigating the nature of the extensions to SLD-Resolution in the ET^* and ET_{interp} algorithms, OLDT-Resolution, SLDAL-Resolution, and the RQA/FQI procedure, we showed that the Backward Fixpoint Procedure formalizes these methods as well. Finally, we argued that an efficient implementation of a resolution-based procedure has to explicitly rely on a semi-naive query evaluator and on the very rewriting of the Alexander and Magic Set methods.

Relying on bottom-up meta-interpreters for specifying fixpoint query answering procedures appears to be a useful technique for both theoretical and practical issues. On the one hand, it often permits simple soundness and completeness proofs, like in this article. On the other hand, bottom-up meta-interpretation can be applied to specifying advanced fixpoint query answering procedures. This technique seems to be an interesting direction for further research.

The Backward Fixpoint Procedure can be called an 'upside-down meta-interpreter', for it relies on bottom-up reasoning for implementing a top-down evaluation. Meta-interpretation can also be applied in the reverse way, i.e., for specifying bottom-up reasoning in a top-down language. We applied this approach for implementing the rather unconventional theorem prover SATCHMO in the top-down language PROLOG [MB 88]. 'Upside-down meta-interpretation' does not seem to have attracted much attention. Further investigations of this technique are desirable.

Finally, efforts should be devoted to investigating strategies for combining top-down and bottom-up reasoning, i.e., strategies for chosing which rules to rewrite à la Alexander/Supplementary Magic Set and which rules to keep unchanged. As recent results in various fields of automated reasoning show, approaches combining the two inference principles often permit considerable gains in efficiency.

Acknowledgements

I am indebted to Jean-Marie Nicolas for his encouragement and support during this research, and to Alexandre Lefebvre and Rainer Manthey for helpful discussions.

References

[BAR 86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM-SIGMOD Conf. on Management of Data (SIGMOD).* Washington, D.C., 1986.

[BEE 89] C. Beeri. Recursive query processing. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS).* Philadelphia, Penn., 1989. Tutorial.

[BMSU 86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems (PODS).* 1986.

[BR 87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS).* San Diego, Calif., 1987.

[BRY 89] F. Bry. *Query evaluation in recursive databases: Bottom-up and top-down reconciled.* Research Report IR-KB-64, ECRC, 1989.

[DIE 87] S.W. Dietrich. Extension tables: Memo relations in logic programming. In *Proc. Symp. on Logic Programming (SLP).* San Francisco, Calif., 1987.

[DR 86] R. Demolombe and V. Royer. *Evaluation strategies for recursive axioms: A uniform presentation.* Internal Report, ONERA-CERT, Toulouse, France, 1986.

[LLO 87] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, Berlin, New York, 1987. 2nd Ed.

[LV 89] A. Lefebvre and L. Vieille. On deductive query evaluation in the Dedgin* system. In *Proc 1st Int. Conf. on Deductive and Object-Oriented Databases (DOOD).* Kyoto, Japan, 1989. In this book.

[MB 88] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proc. 9th Int. Conf. on Automated Deduction (CADE).* Argonne, Ill., 1988.

[NEJ 87] W. Nejdl. Recursive strategies for answering recursive queries - The RQA/FQI strategy. In *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB).* Brighton, Great Brit., 1987.

[RAM 88] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proc. 5th Int. Conf. and Symp. on Logic Programming (ICLP/SLP).* Seattle, Wash., 1988.

[RLK 86] J. Rohmer, R. Lescœur, and J.-M. Kerisit. The Alexander method, a technique for the processing of recursive axioms in deductive databases. *New Generation Computing* 4(3), 1986.

[SEK 89] H. Seki. On the power of Alexander templates. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS).* Philadelphia, Penn., 1989.

[SES 87] P. Sestoft and H. Søndergaard. A bibliography on partial evaluation. *SIGPLAN Notices* 23(2), 1987.

[SKGB 87] H. Schmidt, W. Kiessling, U. Güntzer, and R. Bayer. Compiling exploratory and goal-directed deduction into sloppy delta-iteration. In *Proc. Symp. on Logic Programming (SLP).* San Francisco, Calif., 1987.

[TAR 55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Jour. of Mathematics* 5, 1955.

[TS 86] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proc. 3rd Int. Conf. on Logic Programming (ICLP).* London, Great. Brit., 1986.

[ULL 85] J. D. Ullman. Implementation of logical query languages for databases. *Trans. on Database Systems* 10(3), 1985.

[ULL 89] J. D. Ullman. Bottom-up beats top-down for Datalog. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS).* Philadelphia, Penn., 1989.

[vEK 76] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Jour. of the ACM* 23(4), 1976.

[VIE 86] L. Vieille. Recursive axioms in deductive databases: The Query-Subquery Approach. In *Proc. 1st Int. Conf. on Expert Database Systems (EDS).* Charleston, New Cal., 1986.

[VIE 87] L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proc. 4th Int. Conf. on Logic Programming (ICLP).* Melbourne, Australia, 1987.

[VIE 89] L. Vieille. Recursive query processing: The power of logic. *Theor. Comp. Sc.*, 1989. to appear.