Data & Knowledge Engineering 5 (1990) 289–312
North-Holland

Query evaluation in recursive databases: bottom-up and top-down reconciled

François BRY
ECRC, Arabellastr. 17, D-8000 Munich 81, FRG
e-mail: fb @ ecrc.de

Abstract. 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 processing – 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 the methods based on resolution on the other hand, implement the Backward Fixpoint Procedure. Second, we interpret the rewritings of the Alexander and Magic Set methods as specializations of the Backward Fixpoint Procedure. Finally, we argue that such a rewriting is also needed in a database context 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.

Keywords. Deductive databases, Logic programming, Query answering, Recursive queries, Recursive logic programs, Bottom-up reasoning, Top-down reasoning, Meta-interpretation, Partial evaluation.

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 [34]. 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 [3]. 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.

Many studies have been devoted to fixpoint computations for querying databases. In particular, various search strategies for the semi-naive method are investigated in [25]. The articles [1] and [35] define a fixpoint semantics for a class of non-Horn deductive databases, the class of stratified databases. The variant methods Alexander [24] and Magic Set [2, 5] permit an efficient fixpoint processing of recursive queries on Horn databases by relying on a rewriting of the deduction rules. In [6], we extended these methods to a class of non-Horn databases by using a 'Conditional Fixpoint Procedure'. This fixpoint procedure is extended to a ternary logic in [A] for querying unrestricted non-Horn databases. All these studies rely on naive or semi-naive fixpoint computations.

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 [24] and Magic Set [2, 5] 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. Although unusual in databases, it is implicit in database systems that store predefined queries. As we show below, bottom-up meta-interpretation permits one to obtain a surprisingly simple specification for the Backward Fixpoint Procedure. This approach can also be applied to specify other advanced fixpoint procedures like top-down meta-interpretation is conventionally applied in logic programming for enhancing the interpreter – see, e.g. [28]. Bottom-up meta-interpretation gives rise for example to specifying fixpoint procedures for querying databases with uncertain values or for performing updates specified on derived relations [B].

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 [24] and Magic Set [2, 5] methods are based on this principle. The second approach is an extension of SLD-Resolution [13, 18] that consists of storing the encountered queries and the proven answers. The ET* and ET_{interp} algorithms [11], OLDT-Resolution [30], QSQ and SLDAL-Resolution [37], and the RQA/FQI strategy [22], 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 [5] that the same strategies – called 'sideway information passing strategies' – can be applied to optimize both types of methods. Moreover, Ramakrishnan noticed in [23] that the same propagation of constants is possible with rewriting-based and resolution-based methods. This point was investigated more formally by Ullman in [33]. Commonalities in the inferences of both types of methods were often cited – e.g. in [10, 4, 38]. Recently, Seki established a one-to-one mapping between the inferences performed by methods of both types [26]. 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 – see e.g. [27]. 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 [17]. 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 simple soundness and completeness proofs for the Alexander and Magic Set methods on the one hand, and for the ET* and ET_{interp} algorithms, OLDT-Resolution, QSQ and SLDAL-Resolution, and the RQA/FQI strategy, on the other hand. 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 query answering 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.

The results established in this article have been informally presented in a tutorial on deductive databases during the 6th International Conference on Logic Programming [9]. They have been presented in a shortened form at the 1st International Conference on Deductive and Object-Oriented Databases [7].

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:

$$H \leftarrow L_1 \wedge \cdots \wedge L_n$$

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

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

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 $L_1 \wedge \cdots \wedge L_n$ 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 $H \leftarrow L_1 \wedge \cdots \wedge L_n \in \mathrm{DB}$
- a substitution σ

such that:

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

The $L_i\sigma$ are called *premises* of A.

The *immediate consequence operator* T on DB – formally, on $\mathrm{HB(DB)} \cup \mathrm{DR(DB)}$ – is the function associating with each $\mathrm{D} \subseteq \mathrm{H(DB)} \cup \mathrm{DR(DB)}$ the set $T(\mathrm{D})$ 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:

$$\forall P_1 \subseteq S \qquad \forall P_2 \subseteq S \quad [P_1 \subseteq P_2 \Rightarrow \Gamma(P_1) \subseteq \Gamma(P_2)]$$

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:

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

where:

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

$$\Gamma\uparrow^{n+1}(P) = \Gamma(\Gamma\uparrow^{n}(P)) \cup \Gamma\uparrow^{n}(P) \quad \text{for } n \in \mathbb{N}$$

Intuitively, $T\uparrow^{1}(\mathrm{DB})$ denotes DB augmented with its immediate consequences; $T\uparrow^{\omega}(\mathrm{DB})$ denotes DB augmented with all the facts that can be derived from DB.

A *least fixpoint* of an operator Γ on a set S is a set $\Gamma\uparrow^{n}(S)$ $(n \in \mathbb{N}^* \cup \{\omega\})$ 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 [31]. Therefore, T admits a unique least fixpoint on Horn databases. This fixpoint is finite if $T\uparrow^{\omega}(\mathrm{DB}) = T\uparrow^{n}(\mathrm{DB})$ for some $n < \omega$. This is in particular (but not only) 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 $T\uparrow^{\omega}(\mathrm{DB})$.

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

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

Since the least fixpoint $T\uparrow^{\omega}(\mathrm{DB})$ 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 [3], 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 $T\uparrow^{n+1}(\mathrm{DB})$, all immediate consequences of $T\uparrow^{i}(\mathrm{DB})$ for $0 \leq i < n$ are recomputed. Since T is monotonic on Horn databases, it suffices to generate those elements of $T\uparrow^{n+1}(\mathrm{DB})$ that have at least one premise in $T\uparrow^{n}(\mathrm{DB}) \setminus T\uparrow^{n-1}(\mathrm{DB})$. 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 [25]. These strategies depart from the strict breadth-first generation of consequences.

The naive and semi-naive methods are *sound* query answering procedures for Horn databases, i.e., they generate only facts that belong to the least fixpoint $T\uparrow^{\omega}(\mathrm{DB})$ of T on a Horn database DB. They are *complete* query answering procedures for non-recursive and for function-free Horn databases, i.e. they determine all the facts in $T\uparrow^{\omega}(\mathrm{DB})$. They may never terminate when applied on a database such that $T\uparrow^{\omega}(\mathrm{DB})$ is infinite. Nevertheless, the naive and semi-naive methods are *exhaustive* query answering procedures, i.e. given a ground fact F such that $F \in T\uparrow^{\omega}(\mathrm{DB})$, they always determine this membership in finite – but indefinite – time, even if $T\uparrow^{\omega}(\mathrm{DB})$ is infinite.

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 similar to the so-called 'Prolog in Prolog' or 'vanilla' Prolog meta-interpreter [28]. We show how the fixpoint computation of immediate consequences can be specified by meta-rules intended for bottom-up evaluation. This technique is used in more interesting ways in Section 4.

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

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

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 different evaluation of the body of the above-defined meta-rule would be very inefficient or could compromise termination, for the variable B would have to be bound to all possible atomic or conjunctive expressions.

A bottom-up evaluation of the above-defined rule produces an expression 'fact(F)' for each $F \in T(\mathrm{DB})$. By iterating in the naive or semi-naive manner, one generates an expression 'fact(F)' for each $F \in T\uparrow^{\omega}(\mathrm{DB})$. *Fig. 1* (next page) illustrates this principle on an example. 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 $\sigma_1 = [x{:}a]$ and $\sigma_2 = [x{:}b]$, 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 $\sigma_3 = [x{:}a]$: 'fact($p(a)$)' is proven. The procedure stops because the most recently derived fact $p(a)$ cannot serve as a premise in any rule.

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

$s(b)$ $q(x) \leftarrow s(x)$

Successful derivations:

Step 1: $\text{fact}(q(x)) \leftarrow \text{rule}(q(x) \leftarrow s(x)) \wedge \text{evaluate}(s(x))$ $\sigma_1 = [x{:}a]$, $\sigma_2 = [x{:}b]$

Step 2: $\text{fact}(p(x)) \leftarrow \text{rule}(p(x) \leftarrow q(x) \wedge r(x)) \wedge \text{evaluate}(q(x) \wedge r(x))$ $\sigma_3 = [x{:}a]$

Fig. 1. Bottom-up reasoning with a bottom-up meta-interpreter.

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$(B)\sigma$' 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. Formally, the predicate 'evaluate' could be omitted and 'evaluate(B)' could be replaced by B. In the sequel, in particular in proofs, we shall rely implicitly on this semantics. 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 – see e.g. [28]. It is natural in these languages because they give the same structure to data and programs. More generally, the ability to specify a given programming language in itself is generally considered as a necessary feature of powerful languages.

It is worth noting that the meta-interpreters considered in logic programming are usually intended for top-down evaluation. In contrast, the above-defined meta-interpreter corresponding to the immediate consequence operator is intended for bottom-up processing. Meta-interpreters of the respective types are not interchangeable. Processing the above-defined meta-interpreter with SLD-Resolution would enter an infinite loop in case the object program – i.e. the database under consideration – is recursive. Similarly, conjunctions of unbounded growing lengths are usually generated by processing bottom-up meta-interpreters intended for top-down evaluation. The study of similarities and differences of both types of meta-interpreters seems to be an interesting direction of research. Moreover, bottom-up meta-interpretation seems an interesting technique to investigate, especially for databases.

The variables 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.

Formally, the specification of a query procedure by means of rules can be viewed as extending the first-order, one-sorted language of the database into a first-order, two-sorted language. We do not discuss this issue here – see e.g. [14, 29]. Other formalizations of meta-interpretation rely on second-order logic. Extending a database language with variables ranging over queries is implicitly done in conventional database systems that store predefined queries.

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

Proposition 3.1. *Let* DB *be a Horn database, A a fact, and* $n \in \mathbb{N}^*$.

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

Proof. We first prove by induction on n that:

$$(\dagger) \quad \forall n \in \mathbb{N} \qquad A \in T\uparrow^n(\mathrm{DB}) \Leftrightarrow \mathrm{fact}(A) \in T\uparrow^n(M_{\mathrm{DB}})$$

This is the case for $n = 0$, by definition of M_{DB} and $T\uparrow^0$. Assume that this is true for all k such that $0 \leqslant k < n$. If $A \in T\uparrow^n(\mathrm{DB})$ ($\mathrm{fact}(A) \in T\uparrow^n(M_{\mathrm{DB}})$, resp.), then, by definition of T, there is a rule $H \leftarrow L_1 \wedge \cdots \wedge L_n$ in DB (a fact $\mathrm{rule}(H \leftarrow L_1 \wedge \cdots \wedge L_n)$ in M_{DB}, resp.) and a substitution τ such that $A = H\tau$ ($\mathrm{fact}(A) = \mathrm{fact}(H)\tau$, resp.) and $L_i\tau \in T\uparrow^{n-1}(\mathrm{DB})$ ($\mathrm{fact}(L_i)\tau \in T\uparrow^{n-1}(\mathrm{DB})$, resp.) for all $i = 1, \ldots, n$. By definition of M_{DB}, $\mathrm{rule}(H \leftarrow L_1 \wedge \cdots \wedge L_n) \in M_{\mathrm{DB}}$ ($H \leftarrow L_1 \wedge \cdots \wedge L_n \in \mathrm{DR}(\mathrm{DB})$, resp.), and by induction hypothesis $\mathrm{fact}(L_i)\tau \in T\uparrow^{n-1}(M_{\mathrm{DB}})$ ($L_i\tau \in T\uparrow^{n-1}(\mathrm{DB})$, resp.) for all $i = 1, \ldots, n$. It follows from the definition of T that $\mathrm{fact}(H)\tau \in T\uparrow^n(M_{\mathrm{DB}})$ ($H\tau \in T\uparrow^n(\mathrm{DB})$, resp.). Point 1 follows from ($\dagger$) and from the definition of a least fixpoint. Point 2 is an immediate consequence of ($\dagger$). □

Intuitively, the second point of Proposition 3.1 means that the semi-naive computation of $T\uparrow^\omega(M_{\mathrm{DB}})$ expresses the semi-naive computation of $T\uparrow^\omega(\mathrm{DB})$ 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 – e.g. for test purposes.

Specifying fixpoint 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 – or nested terms – 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.

We conclude this section with a generalization of Proposition 3.1. It formally justifies the use of meta-interpretation with bottom-up rules for specifying fixpoint query answering procedures.

Proposition 3.2. *Let* DB *be a Horn database. Let* MR *be a set of meta-rules defining a predicate 'fact' such that the predicates in* MR *do not occur in* DB *and are defined in terms of the base relations 'rule' and 'fact' that describe* DB. *Let* T_{MR} *be the operator on* DB *which is specified by* MR.
For all $\mathrm{D} \subseteq \mathrm{DB}$ *and* $n \in \mathbb{N} \cup \{\omega\}$, *let*:

$$F_{\mathrm{D}} = \{\mathrm{fact}(F) \mid F \in \mathrm{F}(\mathrm{D})\} \cup \{\mathrm{rule}(R) \mid R \in \mathrm{DR}(\mathrm{D})\}$$

$$S^n_{\mathrm{D}} = D \cup \{A \mid \mathrm{fact}(A) \in T\uparrow^n(F_{\mathrm{D}} \cup \mathrm{MR})\}$$

The following property holds:

$$\forall D \subseteq \mathrm{DB} \qquad \forall n \in \mathbb{N} \cup \{\omega\} \qquad S^n_{\mathrm{D}} = T_{\mathrm{MR}}\uparrow^n(D)$$

Proof. By induction on n, one first establishes the property for $n < \omega$ in the same ways as ($\dagger$) is established in the proof of Propositions 3.1. The property for $n = \omega$ is a consequence of that result and of the definition of a least fixpoint. □

Intuitively, Proposition 3.2 shows that the least fixpoint of an operator T_{MR} can be computed by running its rule-based specification with an evaluator for the immediate consequence operator T. Moreover, if the evaluator for T implements a semi-naive method, this would reflect the semi-naive computation of the least fixpoint for T_{MR} in the meta-language of MR.

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. We define a bottom-up meta-interpreter that 'reverses' the reasoning principle: Processing it bottom-up performs a top-down evaluation at the object level.

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

(ii) $\text{query}_b(B) \leftarrow \text{query}_b(Q) \wedge \text{rule}(Q \leftarrow B)$

(iii) $\text{query}_b(Q_1) \leftarrow \text{query}_b(Q_1 \wedge Q_2)$

(iv) $\text{query}_b(Q_2) \leftarrow \text{query}_b(Q_1 \wedge Q_2) \wedge \text{evaluate}(Q_1)$

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 emphasize that a bottom-up processing of the meta-interpreter defined by rules (i)-(iv) realizes a top-down evaluation of the database rules. In other words, the rules given above implement a top-down evaluation of database rules in a meta-language of bottom-up rules. 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 $T_b\uparrow^{\omega}(\text{DB} \cup \text{Q})$ of the operator T_b on DB and Q. The atoms in Q are the initial queries posed to the database DB. *Fig. 2* shows on an example how the Backward Fixpoint Procedure computes $T_b(\text{DB} \cup \text{Q})$. Note that no t facts are derived.

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

Queries: $\text{query}_b(p(b))$ $\text{query}_b(q(x))$

Successful derivation:

$\text{fact}(q(x)) \leftarrow \text{query}_b(q(x)) \wedge \text{rule}(q(x) \leftarrow s(x)) \wedge \text{evaluate}(s(x))$ $\sigma_1 = [x{:}a]$, $\sigma_2 = [x{:}b]$

$\text{query}_b(q(x) \wedge r(x)) \leftarrow \text{query}_b(p(b)) \wedge \text{rule}(p(x) \leftarrow q(x) \wedge r(x))$ $\sigma_3 = [x{:}b]$

$\text{query}_b(s(x)) \leftarrow \text{query}_b(q(x)) \wedge \text{rule}(q(x) \leftarrow s(x))$ $\sigma_4 = [\ ]$

Fig. 2. Top-down reasoning with the Backward Fixpoint Procedure.

Evaluating the body of rule (i) first binds '$\text{query}_b(Q)$' to '$\text{query}_b(p(b))$', 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. '$\text{query}_b(Q)$' from rule (i) is then bound to '$\text{query}_b(q(x))$', and B to '$s(x)$'. The evaluation of B over the database facts yields the bindings $\sigma_1 = [x{:}a]$ and $\sigma_2 = [x{:}b]$, thus generating '$\text{fact}(q(a))$' and '$\text{fact}(q(b))$'. Rule (ii) generates from '$\text{query}_b(p(b))$' the expression '$\text{query}_b(q(x) \wedge r(x))$' with the binding $\sigma_3 = [x{:}b]$. Similarly, '$\text{query}_b(s(x))$' is derived by rule (ii) from '$\text{query}_b(q(x))$'.

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 '$\text{evaluate}(Q_1)$' 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

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

before '$\text{query}_b(Q)$' in rule (i) would be inefficient because useless '$\text{evaluate}(B)$' 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 '$\text{query}_b(Q)$'.

It is worth noting that, although based on backward reasoning like Linear Resolution [20, 16], the Backward Fixpoint Procedure differs significantly from this method and from procedures related to it, like Model-Elimination [19] and SLD-Resolution [13, 18]. 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

$$\text{query}_b(p(x) \wedge q(x, y))$$

can be generated during the computation of $T{\uparrow}^n(\text{DB})$ 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 '$\text{query}_b(q(a, y))$' during the computation of $T{\uparrow}^{m+1}(\text{DB})$. In contrast, SLD-Resolution would have to recompute the expression

$$\text{query}_b(p(x) \wedge q(x, y))$$

in order to generate '$\text{query}_b(q(a, y))$' 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.*

$$A\tau \in T{\uparrow}^{\omega}(\text{DB}) \quad \textit{iff} \quad \text{fact}(A)\tau \in T_b{\uparrow}^{\omega}(\text{DB} \cup \{\text{query}_b(A)\})$$

The proof of Proposition 4.1 can be sketched as follows. A proof P of $\text{fact}(A)\tau$ in $\text{DB} \cup \{\text{query}_b(A)\}$ yields a proof of $A\tau$ in DB by pruning the query_b-facts from P. Conversely, a proof of $\text{fact}(A)\tau$ in $\text{DB} \cup \{\text{query}_b(A)\}$ is obtained from a proof P of $A\tau$ in DB by inserting query_b-facts in P according to the rules that specify the Backward Fixpoint Procedure.

Proof.

Necessary condition

We first prove by induction on n that for all atoms F, substitutions τ, and integers n, the following implication holds:

$$(\dagger) \quad F\tau \in T\uparrow^n(\mathrm{DB}) \Rightarrow \exists m \in \mathbb{N}\ \mathrm{fact}(F)\tau \in T_b\uparrow^{n+m}(\mathrm{DB} \cup \{\mathrm{query}_b(F)\})$$

Since $T\uparrow^0(\mathrm{DB}) = T_b\uparrow^0(\mathrm{DB}) = \mathrm{DB}$, $(\dagger)$ holds for $n = 0$ with $m = 0$. Let $n \in \mathbb{N}^*$ and assume that $(\dagger)$ holds for all k such that $0 \leq k < n$. Let $F\tau \in T\uparrow^n(\mathrm{DB}) \backslash T\uparrow^{n-1}(\mathrm{DB})$. There is a rule $H \leftarrow L_1 \wedge \cdots L_p$ in DB and a substitution σ such that $F\tau = H\sigma$, and $L_i\sigma \in T\uparrow^{n-1}(\mathrm{DB})$, for all $i = 1, \ldots, p$. By induction hypothesis, there are p integers m_i such that $\mathrm{fact}(L_i)\sigma \in T_b\uparrow^{n-1+m_i}(\mathrm{DB} \cup \{\mathrm{query}_b(L_i)\})$.

Since $\mathrm{query}_b(L_i) \in T_b\uparrow^p(\mathrm{DB} \cup \{\mathrm{query}_b(F)\})$, we have $\mathrm{fact}(L_i)\tau \in T_b\uparrow^{n-1+m+p}(\mathrm{DB} \cup \{\mathrm{query}_b(F)\})$, where m is the maximum of the m_is. Therefore, $(\dagger)$ holds for n. It follows that, for all atoms F and substitutions τ:

$$F\tau \in T\uparrow^\omega(\mathrm{DB}) \Rightarrow \mathrm{fact}(F)\tau \in T_b\uparrow^\omega(\mathrm{DB} \cup \{\mathrm{query}_b(F)\})$$

Sufficient condition

We prove by induction on n that for all atoms F, substitutions τ, and integers n:

$$(\ddagger) \quad \mathrm{fact}(F)\tau \in T_b\uparrow^n(\mathrm{DB} \cup \{\mathrm{query}_b(F)\}) \Rightarrow F\tau \in T\uparrow^n(\mathrm{DB})$$

The property holds for $n = 0$. Let $n \in \mathbb{N}^*$ and assume that $(\ddagger)$ holds for all k such that $0 \leq k < n$. Assume:

$$\mathrm{fact}(F)\tau \in T_b\uparrow^n(\mathrm{DB} \cup \{\mathrm{query}_b(F)\}) \backslash T_b\uparrow^{n-1}(\mathrm{DB} \cup \{\mathrm{query}_b(F)\})$$

By Proposition 3.2, $Fr \in T\uparrow^n(\mathrm{B}) \backslash T\uparrow^{n-1}(\mathrm{B})$, where B is the database with set of facts:

$$\{\mathrm{fact}(A) \mid A \in \mathrm{F}(\mathrm{DB})\} \cup \{\mathrm{query}_b(F)\} \cup \{\mathrm{rule}(A) \mid A \in \mathrm{R}(\mathrm{DB})\}$$

and with rules the set of rules R that specify the Backward Fixpoint Procedure.

There is a rule $r = H \leftarrow L_1 \wedge \cdots \wedge L_p$ in R and a substitution σ such that $\mathrm{fact}(F)\tau = H\sigma$, and $L_i\sigma \in T\uparrow^{n-1}(\mathrm{B})$ for all $i = 1, \ldots, p$. The only rule in R the head predicate of which is 'fact' is rule (i) $\mathrm{fact}(X) \leftarrow \mathrm{query}_b(X) \wedge \mathrm{rule}(X \leftarrow Y) \wedge \mathrm{evaluate}(Y)$.

Hence, there is a rule $A \leftarrow B_1 \wedge \cdots \wedge B_k$ in DB, and a substitution v such that $F\tau = Av$ and $\mathrm{fact}(B_i v) \in T_b\uparrow^{n-1}(\mathrm{DB} \cup \{\mathrm{query}_b(F)\})$, for all $i = 1, \ldots, k$. By induction hypothesis, $B_i v \in T\uparrow^{n-1}(\mathrm{DB})$, for all $i = 1, \ldots, k$. Hence, $F\tau \in T\uparrow^n(\mathrm{DB})$, i.e. $(\ddagger)$ holds for n. It follows that, for all atoms F and substitutions τ:

$$\mathrm{fact}(F)\tau \in T_b\uparrow^\omega(\mathrm{DB} \cup \{\mathrm{query}_b(F)\}) \Rightarrow F\tau \in T\uparrow^\omega(\mathrm{DB})$$

The proof is complete. □

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, complete, and exhaustive query answering method for (possibly recursive) Horn databases.*

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

It is often desirable to evaluate queries with a top-down reasoning method in order to propagate the constants – i.e. in terms of relational algebra, the selections – from the initial and subsequent queries to the queries on the base relations. This can only be achieved by top-down reasoning, indeed.

In the next sections, we first refine the definition of the Backward Fixpoint Procedure. Then we investigate the relationship between this procedure and methods that were proposed for querying recursive databases. We show that the methods based on rewritings of the database rules as well as the methods based on resolution implement the Backward Fixpoint Procedure. In other terms, the top-down reasoning principle is conveyed by the rewriting of these methods.

5. The Backward Fixpoint Procedure revisited

A direct implementation of the 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:

$$\text{fact}(p) \leftarrow \text{query}_b(p) \wedge \text{rule}(p \leftarrow q \wedge r) \wedge \text{evaluate}(q \wedge r) \quad \textit{from (i)}$$

$$\text{query}_b(q \wedge r) \leftarrow \text{query}_b(p) \wedge \text{rule}(p \leftarrow q \wedge r) \quad \textit{from (ii)}$$

$$\text{query}_b(q) \leftarrow \text{query}_b(q \wedge r) \quad \textit{from (iii)}$$

$$\text{query}_b(r) \leftarrow \text{query}_b(q \wedge r) \wedge \text{evaluate}(q) \quad \textit{from (iv)}$$

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($\emptyset$, Q)' denotes a completely non-evaluated query 'Q'. In contrast, 'evaluate(B)' in rules (i) ('evaluate(Q_1)' in rule (iv), resp.) must be replaced by 'evaluate($B, \emptyset$)' ('evaluate($Q_1, \emptyset$)', resp.) which denotes a completed evaluation of B (Q_1, resp.). The following bottom-up rules specify the binary predicate 'evaluate':

(v) $\text{evaluate}(\emptyset, B) \leftarrow \text{query}_b(Q) \wedge \text{rule}(Q \leftarrow B)$

(vi) $\text{evaluate}(B_1, B_2) \leftarrow \text{evaluate}(\emptyset, B_1 \wedge B_2) \wedge \text{fact}(B_1)$

(vii) $\text{evaluate}(B_1 \wedge B_2, B_3) \leftarrow \text{evaluate}(B_1, B_2 \wedge B_3) \wedge B_1 \neq \emptyset \wedge \text{fact}(B_2)$

(viii) $\text{evaluate}(B, \emptyset) \leftarrow \text{fact}(B)$

(ix) $\text{evaluate}(B_1 \wedge B_2, \emptyset) \leftarrow \text{evaluate}(B_1, B_2) \wedge B_1 \neq \emptyset \wedge \text{fact}(B_2)$

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($X, \emptyset$)'.

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

$$\exists B_1 \exists B_3 \ \text{evaluate}(B_1, B_2 \wedge B_3) \in T_b'\uparrow^{\omega}(\text{DB} \cup \text{Q}) \quad \textit{iff} \quad \text{query}_b(B_2) \in T_b'\uparrow^{\omega}(\text{DB} \cup \text{Q})$$

Proof. One first shows by induction on n that, for all integers n:

$$\text{evaluate}(B_1, B_2 \wedge B_3) \in T_b'\uparrow^{n}(\text{DB} \cup \text{Q}) \Rightarrow \exists m \in \mathbb{N} \ \text{query}_b(B_2) \in T_b'\uparrow^{n+m}(\text{DB} \cup \text{Q})$$

By definition of a least fixpoint, we have:

$$\text{evaluate}(B_1, B_2 \wedge B_3) \in T_b'\uparrow^{\omega}(\text{DB} \cup \text{Q}) \Rightarrow \text{query}_b(B_2) \in T_b'\uparrow^{\omega}(\text{DB} \cup \text{Q})$$

Conversely, one proves by induction on n that, for all $n \in \mathbb{N}$:

$$\text{query}_b(B_2) \in T_b'\uparrow^{n}(\text{DB} \cup \text{Q}) \Rightarrow$$
$$\exists \text{m} \in \mathbb{N} \ \exists B_2 \ \exists B_3 \ \text{evaluate}(B_1, B_2 \wedge B_3) \in T_b'\uparrow^{n+m}(\text{DB} \cup \text{Q})$$

It follows that

$$\text{query}_b(B_2) \in T_b'\uparrow^{\omega}(\text{DB} \cup \text{Q}) \Rightarrow \text{evaluate}(B_1, B_2 \wedge B_3) \in T_b'\uparrow^{\omega}(\text{DB} \cup \text{Q})$$

This implication completes the proof. □

By Proposition 5.1, rules (ii)–(iv) can be replaced by the following rules, without affecting the semantics of the operator T_b'.

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

(xi) $\text{query}_b(B_2) \leftarrow \text{evaluate}(B_1, B_2 \wedge B_3)$

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 and atom, and* τ *a substitution such that* $A\tau$ *is ground*

$$\text{evaluate}(A\tau, \emptyset) \in T_b'\uparrow^{\omega}(\text{DB} \cup \text{Q}) \quad \textit{iff} \quad \text{fact}(A\tau) \in T_b\uparrow^{\omega}(\text{DB} \cup \text{Q})$$

Proof. One first proves by induction on n that for all atoms F, substitutions τ, and integers n, the following implication holds:

$$\text{evaluate}(F\tau, \emptyset) \in T_b'\uparrow^{n}(\text{DB} \cup \text{Q}) \Rightarrow \exists m \in \mathbb{N} \ \text{fact}(F\tau) \in T_b\uparrow^{n+m}(\text{DB} \cup \text{Q})$$

in the same way as (†) is established in the proof of Proposition 4.1. Conversely, one proves by induction on n that:

$$\text{fact}(F\tau) \in T_b\uparrow^n(\text{DB} \cup \text{Q}) \Rightarrow \text{evaluate}(F\tau, \emptyset) \in T'_b\uparrow^n(\text{DB} \cup \text{Q})$$

for all atoms F, substitutions τ, and integers n like (‡) is established in the proof of Proposition 4.1. Proposition 5.2 follows by definition of a least fixpoint. □

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 '$\text{query}_b(p(a, b))$' that are not in first normal form, i.e. that contain nested terms. On the other hand, it generates non-ground tuples such as '$\text{query}_b(p(x, b))$'.

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:

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

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

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

which can be simplified into:

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

The expression '$\text{query}_b(p(x))$' can similarly be normalized by specializing the predicate 'query_b' with respect to the relation 'p' into a predicate '$\text{query}_b\text{-}p$':

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

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 [27].

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 $\text{N}(\text{BFP}_{\text{DB}})$ denote the set of

rules and facts obtained from $\mathrm{BFP_{DB}} \cup \mathrm{Q}$ by applying the following rewriting rules, where p denotes a database predicate and $\vec{x}$ a list of terms:

$$\begin{aligned} &\mathrm{fact}(F) &&\rightarrow F \\ &\mathrm{evaluate}(B) &&\rightarrow B \\ &\mathrm{query_b}(p(\vec{x})) &&\rightarrow \mathit{query_b}\text{-}p(\vec{x}) \end{aligned}$$

Lemma 6.1. *Let* DB *be a database,* Q *a set of* $\mathrm{query_b}$*-atoms, and* $n \in \mathbb{N} \cup \{\omega\}$.

1. $\mathrm{fact}(p(\vec{x})) \in T_b\uparrow^n(\mathrm{DB})$ *iff* $p(\vec{x}) \in T\uparrow^n(\mathrm{N}(\mathrm{BFP_{DB}}))$

2. $\mathrm{query_b}(p(\vec{x})) \in T_b\uparrow^n(\mathrm{DB})$ *iff* $\mathit{query_b}\text{-}p(\vec{x}) \in T\uparrow^n(\mathrm{N}(\mathrm{BFP_{DB}}))$

Proof. Let BFP denote the set of rules that specify the Backward Fixpoint Procedure. Lemma 6.1 follows from the fact that the transformation N induces a one-to-one mapping between interpretations of $\{\mathrm{fact}(F) \mid F \in \mathrm{F(DB)}\} \cup \{\mathrm{rule}(R) \mid R \in \mathrm{R(DB)}\} \cup \mathrm{Q} \cup \mathrm{BFP_{DB}}$ and interpretations of $\mathrm{N}(\mathrm{BFP_{DB}})$. □

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 '$\mathrm{query_b}(p(x))$' and '$\mathrm{query_b}(p(y))$' 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 expression 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.*

Proof. If B subsumes A, then by definition there exists a substitution τ of variables in B for terms in A such that $B\tau = A$. Hence, a faithful encoding of A is an instance of B. The necessary condition follows. Conversely, if A and B_c match, then there exists a substitution τ such that $B_c\tau = A\tau$. Since B_c is ground, we have $B_c = A\tau$. The sufficient condition follows from the fact that $B_c = B\sigma$, where σ instantiates variables in B with constants that do not occur in A. □

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 [33].

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 [32], 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 'rv' 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':

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

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' [16]. By Corollary 4.1 this optimization is not necessary for the correctness, the completeness, the exhaustivity, 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:

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

where $n \in \mathbb{N}^*$, 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 body of rule (k) and to beginning subparts of it:

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

We shall denote such a beginning 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) $p(\vec{x}_0) \leftarrow \text{query}_b\text{-}p(\vec{x}_0) \wedge \text{evaluate}(k, n, \vec{x})$ *from* (i)

(b^k) $\text{evaluate}(k, 0, \vec{x}) \leftarrow \text{query}_b\text{-}p(\vec{x}_0)$ *from* (v)

For $j = 0, \ldots, n-1$:

(c_j^k) $\text{evaluate}(k, j+1, \vec{x}) \leftarrow \text{evaluate}(k, j, \vec{x}) \wedge q_{j+1}(\vec{x}_{j+1})$ *from* (vi)–(ix)

(d_j^k) $\text{query}_b\text{-}q_{j+1}(\vec{x}_{j+1}) \leftarrow \text{evaluate}(k, j, \vec{x})$ *from* (x)–(xi)

Proof. Proposition 6.2 is obtained by evaluating over (k) the rule expressions in rules (i)–(ix) and by denoting (k, j) an expression $\wedge_{m=1}^{m=j} q_m(\vec{x}_m)$. □

Fig. 3 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 (a^k)–$(d_j^k)_{1 \leqslant j \leqslant n}$ of Proposition 6.2 from a database rule

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

is precisely the rewriting procedure of the Alexander and of the Supplementary Magic Set method, the improved version of the Magic Set method given in [5]. 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.

Database rules:

(1) $p^b(x) \leftarrow q^b(x) \wedge r(x)$
(2) $q^b(x) \leftarrow s(x)$

Specialized rules:

$p^b(x) \leftarrow \text{query}_b\text{-}p^b(x) \wedge \text{evaluate}(1, 2, x)$

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

$\text{query}_b\text{-}q^b(x) \leftarrow \text{evaluate}(1, 0, x)$

$q^b(x) \leftarrow \text{query}_b\text{-}q^b(x) \wedge \text{evaluate}(2, 1, x)$

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

$\text{query}_b\text{-}s^b(x) \leftarrow \text{evaluate}(2, 0, x)$

Fig. 3. A specialization of the Backward Fixpoint Procedure.

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, complete, and exhaustive query answering methods for (possibly recursive) Horn databases. They are terminating methods for Horn databases defining finitely many facts – e.g. function-free databases.*

The representation of beginning 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 of rules (a^k)-$(d^k_j)_{1 \leq j \leq n}$ 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.

If the rule (k) considered in Proposition 6.2 is adorned, i.e. specialized according to variable instantiation patterns as it is discussed in Section 6.2, then so are rules (a^k)-$(d^k_j)_{1 \leq j \leq n}$. The pre-encoding of variables in expressions '$query_b$-$p(\vec{x}_0)$', '$query_b$-$q_j(\vec{x}_j)$' $(1 \leq j \leq n)$ is realized by removing the variables corresponding to 'f' adornments. If the database rule (k) is not adorned, the adornment can as well be performed on the generated rules (a^k)-$(d^k_j)_{1 \leq j \leq n}$. The result is the same, as it is shown by the following proposition.

Proposition 6.3. *Given a database rule* (k), *let* B(k) *denote the set of rules obtained by specializing with respect to rule* (k) *the rules that specify the Backward Fixpoint Procedure, according to Proposition* 6.2. *Let* A(k) *denote the set of adorned rules associated with rule* (*k*), *according to Section* 6.2. *We have*:

$$A(B(k)) = B(A(k))$$

Proof. Proposition 6.3 follows from the fact that by definition the transformation B does not affect variables, and the transformation A affects only variables occurring in $query_b$-atoms. □

7. From SLD-resolution to fixpoint computation: linearity abandoned

In this section, we consider the algorithms ET* and ET_{interp} [11], OLDT-Resolution [30], QSQ or SLDAL-Resolution [37], and the procedure RQA/FQI [22]. All these methods are based on SLD-Resolution [13, 18] 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.

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 [15]: Algorithm = Logic + Control. In contrast to the Backward Fixpoint Procedure, SLD-Resolution does not share results between different evaluations. Consider the example of *Fig. 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). This feature of SLD-Resolution – and of other methods based on the Resolution principle – is called *linearity*.

The difference between the Backward Fixpoint Procedure and SLD-Resolution 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). In contrast, SLD-Resolution relies on a hierarchical data structure that relates proven facts and generated queries to the queries they come from.

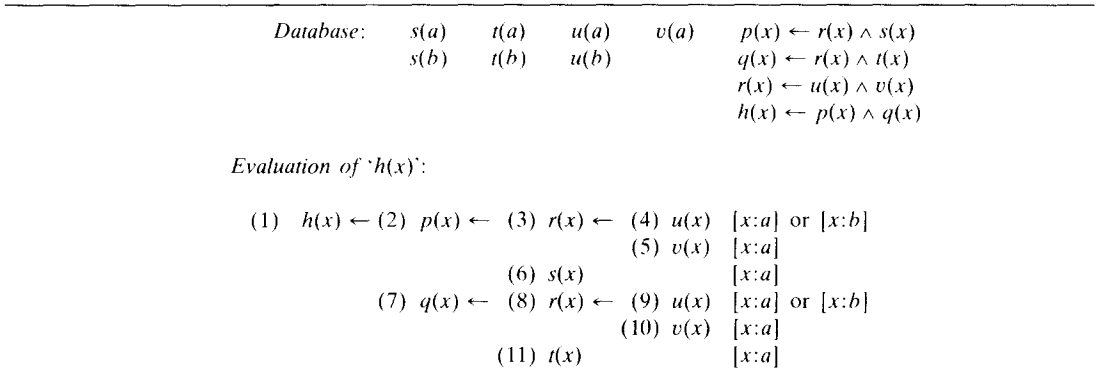

Fig. 4. A SLD-Resolution proof tree.

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:

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

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

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

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

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

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 *Fig. 4*. The ternary predicate 'query_r' associates with a query Q its identifier and the identifiers of the queries it comes from. Thus the two 'r(x)' queries in *Fig. 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). $[I \mid L]$ 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 interpreter, for example, rely on this data structure. In the example of *Fig. 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 [11, 30, 37, 22] 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 definitions 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.

It can also be redundantly specified by means of rules in which the query identifiers are kept. A set-oriented implementation of SLDAL-Resolution is reported in [17]. It relies on a rewriting of the database rules similar to that of the Alexander and Supplementary Magic Set

methods. It processes the rewritten rules with a sort of semi-naive inference engine. This engine, however, uses a hierarchical data structure in main memory that relates the encountered queries to the queries they are issued from. This makes the semi-naive computations more complex.

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, complete, and exhaustive query answering methods for (possibly recursive) Horn databases.*

They are terminating query answering methods for Horn 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 [11] 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 [36] – QSQ is corrected in [22] and [37]. 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 [36] 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 [3]. 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 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 [25].

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 [24], in [5], 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 are exhaustive and 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 [24] and Magic Set [2, 5] 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 [11] with the ET* and $\text{ET}_{\text{interp}}$ algorithms, in [30] with OLDT-Resolution, in [37] with QSQ and SLDAL-Resolution, and in [22] 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 – SLD-Resolution performs top-down reasoning. 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 [5] 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 then the bottom-up reasoning immediate consequence operator of van Emden and Kowalski [34].

We specified a fixpoint query answering procedure, which we call the *Backward Fixing 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. Thus, it is possible to process queries by computing fixpoints without necessarily sticking to the bottom-up reasoning principle of the naive and semi-naive methods. The Backward Fixpoint Procedure was shown to be a sound, complete, and exhaustive query answering method for (possibly recursive) Horn 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. Specialization is a common technique in artificial intelligence [27]. It is used for improving the efficiency of meta-interpreters.

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 Supplementary 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, we have applied bottom-up meta-interpretation for specifying advanced fixpoint query answering procedures, e.g. for databases with uncertain data. 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 [21]. Upside-down meta-interpretation does not seem to have attracted much attention. The article [12] which describes an approach similar to that of the Alexander and Magic Set methods seems to be a noticeable exception.

In [8] we studied this technique by referring to SATCHMO on the one hand, to the Alexander and Magic Set methods on the other hand. We refuted the intuition that direct implementations of a reasoning principle – bottom-up or top-down – necessarily yield better performances than implementing it by means of the other principle.

Further research on upside-down meta-interpretation is desirable. In particular, efforts should be devoted to investigating strategies for combining top-down and bottom-up reasoning, i.e. strategies for choosing 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.

The research reported in this article has been partially supported by the European Community in the framework of the ESPRIT Basic Research Action "Compulog" No. 3012.

References

[1] K.R. Apt, H.A. Blair and A. Walker, *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA 1988) Chapter: Towards a theory of declarative knowledge.

[2] 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. Principles of Database Systems (PODS)* (1986).

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

[4] C. Beeri, Recursive query processing, in: *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS)*, Philadelphia, PA (1989). (Tutorial).

[5] C. Beeri and R. Ramakrishnan, On the power of magic, in: *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS)*, San Diego, CA (1987).

[6] F. Bry, Logic programming as constructivism: A formalization and its application to databases, in: *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS)*, Philadelphia, PA (1989).

[7] F. Bry, Query evaluation in recursive databases: Bottom-up and top-down reconciled in: *Proc. 1st Int. Conf. Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan (1989).

[8] F. Bry, *Upside-down deduction*, Research Report IR-KB-66, ECRC, 1989.

[9] F. Bry and R. Manthey, Deductive Databases, *6th Int. Conf. Logic Programming (ICLP)*, Lisbon, Portugal (1989) (Tutorial).

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

[11] S.W. Dietrich, Extension tables: Memo relations in logic programming, in: *Proc. Symp. Logic Programming (SLP)*, San Francisco, CA (1987).

[12] J. Gallagher, M. Codish and E. Shapiro, Specialisation of Prolog and FCP programs using abstract interpretation, New Generation Comput. 6 (1988).

[13] R. Hill, *LUSH-Resolution and its completeness*, DCL Memo 78, Univ. of Edinburgh, UK, 1974.

[14] P.M. Hill and J.W. Lloyd, Analysis of meta-programs, in: *Proc. Workshop Meta-Programming in Logic Programming*, Bristol, UK (1988).

[15] R. Kowalski, Algorithm = Logic + Control, *Commun. ACM* (1979).

[16] R. Kowalski and D. Kuehner, Linear resolution with selection function, *Artificial Intelligence* 2 (1971).

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

[18] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, New York, 1987), 2nd edition.

[19] D.W. Loveland, A linear format for resolution, in: *Proc. IRIA Symp. Automatic Demonstration*, Versailles, France, (1968), *LNCS* 125 (Springer, Berlin, New York, 1970).

[20] D. Luckham, Refinements theorems in resolution theory, in: *Proc. IRIA Symp. Automatic Demonstration*, Versailles, France, (1968), *LNCS* 125 (Springer, Berlin, New York, 1970).

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

[22] W. Nejdl, Recursive strategies for answering recursive queries – The RQA/FQI strategy, in: *Proc. 13th Int. Conf. Very Large Data Bases (VLDB)*, Brighton, UK (1987).

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

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

[25] H. Schmidt, W. Kiessling, U. Güntzer and R. Bayer, Compiling exploratory and goal-directed deduction into sloppy delta-iteration, in: *Proc. Symp. Logic Programming (SLP)*. San Francisco, CA (1987).

[26] H. Seki, On the power of Alexander templates in: *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS)* Philadelphia, PA (1989).

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

[28] L. Sterling and E. Shapiro, *The Art of Prolog* (MIT Press, Cambridge, MA, 1986).

[29] V.S. Subrahmanian, Foundations of metalogic programming, in: *Proc. Workshop Meta-Programming in Logic Programming*, Bristol, UK (1988).

[30] H. Tamaki and T. Sato, OLD resolution with tabulation, in: *Proc. 3rd Int. Conf. Logic Programming (ICLP)*, London, UK (1986).

[31] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific J. Math.* 5 (1955).

[32] J.D. Ullman, Implementation of logical query language for databases, *Trans. Database Systems* 10(3) (1985).

[33] J.D. Ullman, Bottom-up beats top-down for Datalog, in: *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS)* Philadelphia, PA (1989).

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

[35] A. Van Gelder, *Foundations of deductive databases and logic programming* (Morgan Kaufmann, Los Altos, CA., 1988) Chapter: Negation as failure using tight derivations for general logic programs.

[36] L. Vieille, Recursive axioms in deductive databases: The Query-Subquery approach, in: *Proc. 1st Int. Conf. Expert Database System (EDS)*, Charleston, SC (1986).

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

[38] L. Vieille, Recursive query processing: The power of logic, *Theoret. Comput. Sci.* 69(1) (1989).

[A] F. Bry, Negation in logischer Programmierung: Eine Formalisierung in konstruktiver Logik, in: *Proc. 1. Workshop Informationssysteme und Künstliche Intelligenz*, Ulm, FRG (1990) (Invited paper).

[B] F. Bry, Intensional updates: Abduction via deduction, in: *Proc. 7th Internat. Conf. Logic Programming (ICLP)*, Jerusalem (1990).

François Bry, born 1956, works since 1985 on deductive databases in the Knowledge Base group at ECRC in Munich. His research in this centre has been devoted to database integrity and consistency, query evaluation, semantics of deductive databases and logic programming, and automated theorem proving. He leads a team currently investigating the application of automated reasoning techniques to databases, with the aim to enhance knowledge assimilation. Prior to this, he worked on statistical databases at INRETS, a public research centre in Paris. Bry received his MSc in Mathematics (1978) and his PhD in Graph Theory (1981) from Paris University. He is member of AFCET, ALP, and GI.