in: Proc. 6emes Journees Bases de Donnee
Avancees (BDA'90), Montpellier, Fran
Sept. 25-28, 1990

Upside-down Deduction

François Bry

ECRC, Arabellastr. 17, 8000 München 81, West Germany
fb@ecrc.de

ABSTRACT *Over the recent years, several proposals were made to enhance database systems with automated reasoning. In this article we analyze two such enhancements based on meta-interpretation. We consider on the one hand the theorem prover* Satchmo, *on the other hand the* Alexander *and* Magic Set *methods. Although they achieve different goals and are based on distinct reasoning paradigms, Satchmo and the Alexander or Magic Set methods can be similarly described by* upside-down meta-interpreters, *i.e., meta-interpreters implementing one reasoning principle in terms of the other. Upside-down meta-interpretation gives rise to simple and efficient implementations, but has not been investigated in the past. This article is devoted to studying this technique. We show that it permits one to inherit a search strategy from an inference engine, instead of implementing it, and to combine bottom-up and top-down reasoning. These properties yield an explanation for the efficiency of Satchmo and a justification for the unconventional approach to top-down reasoning of the Alexander and Magic Set methods.*

1. Introduction

During the last years, several proposals were made to enhance database systems by applying artificial intelligence techniques, in particular automated reasoning. One can distinguish two types of extensions. On the one hand, artificial intelligence is applied to improve the implementation of some components of database systems. On the other hand, the database systems themselves are extended with deductive capabilities in order to manage general, non-factual knowledge. Examples of the first type of extensions are, among others, the procedures for improving integrity checking (e.g., [KSS 87], [BDM 88]) and the rule-based approaches to query optimization (e.g., [FRE 87], [GDW 87]). Extensions of the second type are query evaluation procedures for deductive databases like the Alexander [R* 86] and the Magic Set [B* 86, BR 87] methods, or the theorem prover Satchmo [MB 88] which was developed for checking the consistency of integrity constraints [BM 86].

These extensions are not always slavishly imitated from artificial intelligence. Some of them introduce new ideas and techniques that are applicable to automated deduction in general. So do Satchmo, the Alexander, and the Magic Set methods. They rely on meta-interpretation for implementing one reasoning principle – bottom-up or top-down processing – in terms of the other. Although meta-interpretation is a common technique in functional and logic programming, its use for 'reversing' inferences does not seem to have received much attention in the past. We call 'upside-down meta-interpretation' this unconventional use of meta-interpretation.

This paper is devoted to studying upside-down meta-interpretation. We consider two different applications of this technique, in Satchmo on the one hand, in the Alexander and Magic Set methods (short, AMS methods) on the other hand. These applications of upside-down meta-interpretation are symmetrical in many ways. Satchmo performs bottom-up reasoning, the AMS methods compute top-down deductions. Satchmo aims at managing rather small data while the AMS methods are query evaluation procedures for large databases. Being implemented in Prolog, Satchmo relies on the hierarchical data structure classically retained for linear resolution. In contrast, the

AMS methods rely on relational, normalized data structures. Satchmo is implemented in the top-down language Prolog, while the AMS methods rely on a language of rules intended for bottom-up evaluation.

Despite of their differences, Satchmo and the AMS methods benefit from the same properties of upside-down meta-interpretation. They do not re-implement search strategies – depth-first for Satchmo, breadth-first for the AMS methods – but inherit them from the inference engine – Prolog for Satchmo, the semi-naive procedure for the AMS methods. Moreover, Satchmo as well as the AMS methods give rise to combine bottom-up and top-down reasoning during the same process. We rely on the symmetry and complementarity of Satchmo and the AMS methods for investigating these properties of upside-down meta-interpretation.

Satchmo consists of a collection of Prolog programs that are variations on two bottom-up reasoning procedures to be applied in different cases. These programs are very short: Each of them consists of seven to nine Prolog clauses with no more than eight literals. In the article [MB 88], we published them and reported about their good performance on more than eighty benchmark problems recently proposed in the theorem proving literature. In the present article, we study the implementation technique of Satchmo and we submit an explanation for its efficiency.

The Alexander and and the Magic Set methods implement a top-down evaluation of the database rules by means of auxiliary rules that are processed bottom-up. Because they combine bottom-up and top-down deductions in an unconventional manner, they were often misunderstood. In [BRY 89], these methods have been formalized in terms of meta-interpretation. In particular, it has been shown that their auxiliary rules result from the specialization of a meta-interpreter. Here, we rely on the meta-interpretative formalization of the Alexander and Magic Set methods for justifying their 'magic', i.e., the use of bottom-up reasoning for implementing top-down evaluation. We argue that this 'upside-down' approach is preferable to conventional implementations based on linear resolution, e.g., relying on Prolog.

This article consists of seven sections, the first of which is this introduction. We give background notions and notations in Section 2. In Section 3, we briefly recall the meta-interpretation technique. Satchmo is briefly described in Section 4. The meta-interpretative formalization of the Alexander and Magic Set methods is outlined in Section 5. In Section 6 we investigate the advantages of upside-down meta-interpretation. In Section 7 we summarize the article and we indicate open research problems.

2. Background

A deductive database is a finite set of deduction rules and facts. Facts are ground atoms and deduction rules are expressions of the form

$$H \leftarrow L_1 \wedge ... \wedge L_n$$

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

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

where the x_js are the variables occurring in H or in the L_is. H is the *head* of the rule, $L_1 \wedge ... \wedge L_n$ is its *body*. Rules and databases are *function-free* if they contain no function symbols.

In this article, we consider *Horn rules*, i.e., rules such that the L_is are atoms. A database with Horn rules is a *Horn database*. A Horn rule is *safe* or *range-restricted* if each variable occurring in its head also occurs in its body.

A ground atom A is an *immediate consequence* of a database DB if it is derivable from DB by *modus ponens*, i.e., if there exist:

- a rule $H \leftarrow L_1 \wedge ... \wedge L_n \in DB$
- a most general substitution σ

such that:

- $H\sigma = A$
- $L_i\sigma \in DB$ for $i = 1, ..., n$.

Generating immediate consequences is called *forward* or *bottom-up reasoning*.

The *immediate consequence operator* T is the function associating with a database DB the set T(DB) of its immediate consequences. T is monotonic on Horn databases.

Therefore, T has a unique least fixpoint [TAR 55] $T\uparrow^{\omega}(DB)$ on a Horn database DB. We recall that:

$$T\uparrow^{\omega}(DB) = \cup_{n \in N} T\uparrow^{n}(DB)$$

where:

$$T\uparrow^{0}(DB) = DB$$
$$T\uparrow^{n+1}(DB) = T(T\uparrow^{n}(DB)) \cup T\uparrow^{n}(DB) \quad \text{for } n \in N$$

If $T\uparrow^{\omega}(DB)$ is finite – for example if DB is function-free – then there exists $n \in N$ such that:

$$T\uparrow^{\omega}(DB) = T\uparrow^{n}(DB)$$
$$T\uparrow^{k}(DB) \neq T\uparrow^{\omega}(DB) \quad \text{for } k < n$$

The semantics of a Horn database DB is formalized by defining its true facts as the facts in the least fixpoint $T\uparrow^{\omega}(DB)$. They can be obtained by iteratively computing the sets $T\uparrow^{m}(DB)$ for increasing m. This approach to bottom-up reasoning is often called the *naive method.*

Bottom-up reasoning by computing the sets $T\uparrow^{m}(DB)$ leads to redundant computations: While computing $T\uparrow^{m+1}(DB)$, all immediate consequences of $T\uparrow^{i}(DB)$ for $0 \leq i \leq m$ are re-computed. Since T is monotonic on Horn databases, it suffices to generate those elements of $T\uparrow^{m+1}(DB)$ that have at least one premise in $T\uparrow^{m}(DB) \setminus T\uparrow^{m-1}(DB)$. This approach is called the *semi-naive method.*

An alternative to bottom-up reasoning is *backward* or *top-down reasoning.* A top-down evaluation of an atomic query Q consists in generating n subgoals $L_i\sigma$ $(1 \leq i \leq n)$ for each rule $H \leftarrow L_1 \wedge ... \wedge L_n \in DB$ such that σ is a most general unifier of H and Q. A subgoal $L_i\sigma$ in turn similarly induces new subgoals, or evaluates to $(L_i\sigma)\tau$ if there exists a fact $F \in DB$ and a most general substitution τ such that $F = (L_i\sigma)\tau$.

Top-down significantly differs from bottom-up reasoning as it generates two sorts of data, namely facts and goals. Moreover, top-down reasoning makes use of the relationship between a goal and its subgoals for collecting the substitutions inherited from the evaluations of the latter.

Several strategies have been investigated for top-down reasoning. The implementation of Satchmo relies on one of them, the linear, depth-first strategy of SLD-Resolution. This strategy processes one (sub)goal at a time and generates new subgoals from one rule at a time. Choice-points indicate the remaining alternatives.

This strategy gives rise to use a stack for storing the goals, the choice-points, and for expressing the ancestor/descendant relationship between goals. It was independently proposed in [LOV 68] and in [LUC 68] for general clauses. It is especially well-suited to automated theorem proving because an efficient stack management is easily implemented in main memory. The linear, depth-first strategy and the stack data structure have been retained for Prolog interpreters [G* 85].

---

r_1: $p(x) \leftarrow q(x) \wedge r(x)$ $\quad$ f_1: $q(a)$ $\quad$ f_3: $r(b)$
r_2: $p(x) \leftarrow s(x) \wedge t(x)$ $\quad$ f_2: $q(b)$ $\quad$ f_4: $r(c)$

(1) query: p(x)

(2) query: q(x) | query: r(x) | <u>query: p(x)</u> $\rightarrow r_2$

(3) <u>query: q(a)</u> $\rightarrow f_2$ | query: r(a) | <u>query: p(a)</u> $\rightarrow r_2$

(4) <u>query: q(a)</u> $\rightarrow f_2$ | query: r(a) | <u>query: p(a)</u> $\rightarrow r_2$
the query r(a) *fails*

(5) <u>query: q(b)</u> | query: r(b) | <u>query: p(b)</u> $\rightarrow r_2$

(6) <u>query: q(b)</u> | <u>query: r(b)</u> $\rightarrow f_4$ | <u>query: p(b)</u> $\rightarrow r_2$
the query r(b) *succeeds yielding the solution* p(b)

Fig. 1

Figure 1 gives a simplified representation of the evaluation stack during the first steps of a linear top-down evaluation. A goal is annotated 'query' and a choice-point is represented by a pointer to the next rule or fact: This representation assumes that rules and facts are ordered. A goal expanded with a rule is underlined and its subgoals are

pushed in the stack. In this example, we assume a left-to-right selection function, i.e., the leftmost non-underlined goal is processed first.

3. Top-down and Bottom-up Meta-interpreters

A database DB defines a first-order language L(DB) which is defined as follows. The predicates of L(DB) are the relations of DB. Its constants and function symbols are those occurring in the facts or in the rules of DB. Variables in L(DB) range over the terms that can be constructed from the constants and function symbols.

One can also consider formulas in L(DB) as terms. This gives rise to define new predicates expressing properties of these formulas. Such predicates are called *meta-predicates*. For example, the rules of Figure 2 define a meta-predicate 'proven' which defines the conjunctions and facts that are true in the database under consideration.

$$\begin{aligned} \text{proven}(x) &\leftarrow \text{fact}(x) \\ \text{proven}(x) &\leftarrow \text{rule}(x \leftarrow y) \wedge \text{proven}(y) \\ \text{proven}(x_1 \wedge x_2) &\leftarrow \text{proven}(x_1) \wedge \text{proven}(x_2) \end{aligned}$$

Fig. 2

We assume that the 'fact' meta-predicate of Figure 2 ranges over the facts in the database, the 'rule' meta-predicate, over the database rules. Meta-predicates such as 'fact' and 'rule' are implictly used in relational database systems. These systems usually store in auxiliary relations the information about user-defined relations, e.g., names, arities, etc. Such auxiliary relations corresponds in logic to meta-predicates.

The facts that are true in a database DB – i.e., the facts in the fixpoint $T\uparrow^{\omega}(DB)$ – can be generated through the rules of Figure 2. It is worth noting, however, that a bottom-up processing of these rules would never stop, even if $T\uparrow^{\omega}(DB)$ is finite. Indeed, conjunctions with unbounded lengths would be constructed. In contrast, an exhaustive top-down reasoning method would construct $T\uparrow^{\omega}(DB)$ and stop if this fixpoint is finite.

$$\text{fact}(x) \leftarrow \text{rule}(x \leftarrow y) \wedge \text{evaluate}(y)$$

Fig. 3

The rule of Figure 3 is a bottom-up counterpart to Figure 2. Its bottom-up evaluation generates $T\uparrow^{\omega}(DB)$ and stops if this fixpoint is finite. We assume that the 'evaluate' meta-predicate expresses the evaluation of atomic and conjunctive queries against the facts only, without considering the rules. 'evaluate' can be implemented by calling a relational, non-deductive query evaluator.

Rules whose variables range over formulas are called *meta-rules*. If in addition they specify the evaluation of other rules – the *object rules* –, the meta-predicate they define is called a *meta-interpreter*. Meta-interpretation is commonly used in functional and logic programming, for formal specifications as well as for implementations. We shall speak of *bottom-up* and *top-down meta-interpreters* for refering to the reasoning principle which is intended for evaluating the meta-rules.

$$\begin{aligned} \text{proven}(x \text{ since true}) &\leftarrow \text{fact}(x) \\ \text{proven}(x \text{ since } (y \text{ since } z)) &\leftarrow \text{rule}(x \leftarrow y) \\ &\quad \wedge \text{proven}(y \text{ since } z) \\ \text{proven}((x_1 \wedge x_2) \text{ since } (y_1 \wedge y_2)) &\leftarrow \text{proven}(x_1 \text{ since } y_1) \\ &\quad \wedge \text{proven}(x_2 \text{ since } y_2) \end{aligned}$$

Fig. 4

The meta-interpreters of Figures 2 and 3 are not really useful, for they do not achieve more than the considered inference engine and induce an undesirable overhead. However, slight modifications yield interesting meta-

interpreters. Figure 4 gives an example of such a modification of the rules of Figure 2. The new meta-interpreter generates not only the facts that are true in a database, but also the associated proof trees. 'since' denotes an infixed binary function symbol.

Meta-interpretation is usually used like in Figure 4 for refining an inference engine. Conventional meta-interpreters are intended for top-down computation and reflect top-down reasoning on the object rules – see, e.g., [STS 86]. In this paper, we consider meta-interpreters of a different kind, namely meta-interpreters that rely on one inference principle – bottom-up or top-down – for implementing the other.

4. Satchmo: Proving Theorems by Building Databases

Most theorem provers proceed by refutation, i.e., in order to prove that a formula F is a theorem in a theory or set of axioms T, they establish the inconsistency of $T \cup \{\neg F\}$. One approach to refutation consists in trying to build models – more precisely Herbrand models – of the set of formulas to refute. If there are no models, the set is proven inconsistent.

We shall assume that formulas are in clausal form, i.e., they are in prenex conjunctive normal form and existentially quantified variables are represented by Skolem functions. A clause

$$C_1 \vee ... \vee C_n \vee \neg A_1 \vee ... \vee \neg A_m$$

with positive literals C_is will be represented by the extended rule

$$C_1 \vee ... \vee C_n \leftarrow A_1 \wedge ... \wedge A_m$$

Completely positive clauses (m = 0) are represented by

$$C_1 \vee ... \vee C_n \leftarrow \text{true}$$

Similarly, completely negative clauses (n = 0) are expressed as

$$\text{false} \leftarrow A_1 \wedge ... \wedge A_m$$

Thus negation never occurs explicitly.

Every model M of a set of first-order formulas can be represented by the factual database DB(M) consisting of the positive ground literals that are true in M. By the closed world assumption, the negative literals of M are implicit in DB(M).

An extended rule

$$C_1 \vee ... \vee C_n \leftarrow A_1 \wedge ... \wedge A_m$$

is satisfied in a database DB if for every substitution σ such that $[A_1 \wedge ... \wedge A_m]\sigma$ holds in DB, $C_i\sigma \in$ DB for some i. Conversely, this extended rule is violated in DB if there is an instance $[A_1 \wedge ... \wedge A_m]\sigma$ which is true in DB and if none of the $C_i\sigma$ hold in DB.

This suggests to prove consistency by constructing factual databases instead of full models. Consider for example the database of Figure 5 and the following two rules:

1. $p(y) \leftarrow p(x) \wedge q(x, y)$
2. $q(x, z) \leftarrow q(x, y) \wedge q(y, z)$

Rule 1 is satisfied because if its body holds for some values of x and y, then its head holds for the same value of y. In contrast, Rule 2 is violated: The substitution [x:a, y:b, z:a] yields a solution for the body but no solutions for the head. A database expressing a model of the two rules can be constructed by adding the missing fact, namely q(a, a).

---

p(a)	q(a, b)
p(b)	q(b, a)
p(c)	q(b, b)

Fig. 5

This approach extends to non-Horn rules by considering the various ways to satisfy disjunctions. Consider for example the following extended rule:

3. $[q(x, z) \vee r(x, z)] \leftarrow q(x, y) \wedge q(y, z)$

The database of Figure 5 can be extended into a representation of a (minimal) model of Rule 3 in two ways, by inserting either q(a, a) or r(a, a).

The Prolog program of Figure 6 (on next page) performs such a case analysis. It uses the following data structures. A fact is directly stored as a Prolog fact. An extended rule:

$$[C_1 \vee ... \vee C_m] \leftarrow A_1 \wedge ... \wedge A_n$$

is stored in a Prolog binary relation 'rule' as:

rule((C_1 ; ... ; C_m) , (A_1, ..., A_n))

We recall that ';' and ',' are the Prolog notations for $\vee$ and $\wedge$, respectively. The Boolean variable 'true' is a Prolog built-in which is always satisfied.

```
consistent :-
        rule(H, B),
        B, not H, !,
        component(C, H),
        assume(C),
        not false,
        consistent.
consistent.

component(C1, (C1 ; D)).
component(C, (B ; D)) :-
        !, component(C, D).
component(C, C).

assume(A) :-
        asserta(A).
assume(A) :-
        retract(A), !, fail.
```

Fig. 6

The procedure 'consistent' first searches the rules $H \leftarrow B$. When a rule is found such that its body holds in the already constructed database (test 'B'), its head is evaluated (test 'H') over this database. If H is not already satisfied, then one of its components C is determined. The call 'assume(C)' inserts it into the Prolog database. On backtracking, it is removed, and the next component of H is tried. The test 'not false' forces backtracking when 'false' is generated. Otherwise, the recursive call to 'consistent' pursues the database building. Processing the rules for 'consistent' top-down performs a bottom-up evaluation of the rules in the meta-predicate 'rule'.

As soon as no new conclusions can be generated, i.e., when the test

'rule(H, B), B, not H'

fails, the second clause for 'consistent' is evaluated: It succeeds meaning that a database expressing a model has been built. When all alternatives yield to inconsistency, i.e., they all induce the fact 'false', the procedure fails.

The disjunctions generated by bottom-up reasoning on the extended rules are not explicitly stored in the Prolog database. Instead, the program of Figure 6 relies on the Prolog evaluation stack for expressing them. This is achieved by calling the procedures 'component' and 'assume'. Calling 'component(C, D)' with a variable C and with D instantiated to a disjunction successively binds C to the components of D – by the last clause, to D itself if it is an atom. Calling 'assume(A)' with A instantiated to a fact inserts A into the Prolog database. During backtracking, this fact is removed.

The use of Prolog's backtracking for exploring the various alternatives is illustrated on Figure 7. This figure traces the procedure of Figure 6 on an inconsistent set of non-Horn rules. The procedure fails since all alternatives imply false.

$p(a) \leftarrow true$ $\quad$ $t(x) \leftarrow s(x)$

$[q(x) \vee r(x)] \leftarrow p(x)$ $\quad$ $false \leftarrow q(a)$

$[q(x) \vee s(x)] \leftarrow r(x)$ $\quad$ $false \leftarrow p(x) \wedge t(x)$

p(a) since true
q(a) ∨ r(a) since p(a)
– *assumption:* q(a)
false since q(a)
– *assumption:* r(a)
q(a) ∨ s(a) since r(a)
– *assumption:* q(a)
false since q(a)
– *assumption:* s(a)
t(a) since s(a)
false since p(a) ∧ t(a)

Fig. 7

Incorrect assumptions could be generated from rules that are not range-restricted. Consider for example the following extended rules:

a. $p \leftarrow true$ $\quad$ c. $false \leftarrow q(a)$

b. $[q(x) \vee r(x)] \leftarrow p$ $\quad$ d. $false \leftarrow r(b)$

Assuming 'q(x)' or 'r(x)' means in fact assuming '$\forall x$ q(x)' or '$\forall x$ r(x)'. By Rules c and d no models satisfy these universal statements. However, the database {p, r(a), q(b)} is a model. If the considered extended rules are range-restricted, only ground facts and ground disjunctions are generated and the program of Figure 6 makes valid assumptions. Range-restricted versions of general rules can be obtained by relying on an auxiliary predicate expressing the database domain [MB 88].

The procedure of Figure 6 implements the naive method. The article [BDM 88] gives a semi-naive version. However, the overhead resulting from the management of difference sets $T\uparrow^m(DB) \setminus T\uparrow^{m-1}(DB)$ is in theorem proving often greater than the redundant computations it avoids. This is because of the small data considered in theorem proving.

Satchmo is a collection of Prolog programs that are based on the basic procedure of Figure 6. Although this procedure is not a complete theorem prover, it solves a large class of problems with considerable efficiency. Completeness is achieved by a refinement given in [MB 88]. The other programs of Satchmo are not much longer than that of Figure 6. They are surprisingly efficient – see [MB 88]. In Section 6, we submit an explanation for this efficiency which refers to the upside-down meta-interpretative nature of Sachmo.

5. Alexander and Magic Set Methods: Queries as Facts

The semi-naive method is an attractive inference engine for databases. From a theoretical viewpoint, it is very close to the fixpoint semantics of deductive databases. Moreover, it is by essence amenable to set-oriented query processing. In addition, it is a complete procedure for querying recursive databases and always terminates when there are finitely many answers, e.g., in function-free databases. From a practical viewpoint, the semi-naive method is simple and easily implemented by relying on relational, non-deductive query evaluators. The semi-naive method does not require non-first-normal-form data structures like, e.g., the evaluation stack of Prolog.

Despite of these important features, the semi-naive method has a serious drawback: Because it performs bottom-up reasoning, it might compute more intermediate results than necessary. This point is illustrated by Figure 8 which gives two rules defining the transitive closure t of a direct reachability relationship p represented by a graph, an arc $c_1 \rightarrow c_2$ denoting a fact $p(c_1, c_2)$.

In order to answer the query t(a, x), i.e., to compute all the nodes x directly or indirectly reachable from a in the graph of Figure 8, the semi-naive method computes the whole transitive closure t. In particular, it computes the reachability relationships within the connected component containing f, g and h.

In contrast, the subgoals generated during top-down reasoning inherit some restrictions from the initial goal t(a, x). These subgoals are p(a, y), t(b, z), t(d, z), etc. Each generated subgoal refers to a constant in the connected component containing a. Therefore, no reachability relationships between constants in the other connected component are computed.

(i) $t(x, y) \leftarrow p(x, y)$

(ii) $t(x, z) \leftarrow p(x, y) \wedge t(y, z)$

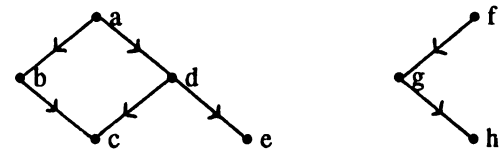

Fig. 8

Intuitively, restrictions present in a query can be used for recursively inducing new restrictions when the proof trees are constructed from the query – i.e., top-down – but not if they are built from the leafs – i.e., bottom-up.

The Alexander and the Magic Set methods (short, AMS methods) retain the advantages of both, the semi-naive method and top-down reasoning. They implement top-

down reasoning by means of auxiliary rules intended for bottom-up processing.

We formalized the AMS methods with upside-down meta-interpretation in [BRY 89] in order to show that they are basicaly identical with the extension of SLD-Resolution proposed under various names – *ET** algorithm in [DIE 87], OLDT-Resolution in [TS 86], QSQ or SLDAL-Resolution in [VIE 87], RQA/FQI strategy in [NEJ 87], etc.

Here, we consider the AMS methods as examples of upside-down meta-interpretation. They are somehow complementary to Satchmo: While Satchmo implements bottom-up reasoning in the top-down language Prolog, the AMS methods specify top-down reasoning by means of bottom-up rules. In the rest of this Section, we informally recall the formalization of the AMS methods as upside-down meta-interpreters.

In contrast with bottom-up reasoning, top-down reasoning generates data of two sorts, facts and queries. The generation of both types of data can be similarly formalized by means of the meta-rules of Figure 9. These rules are intended for bottom-up processing.

As in the program of Figure 6, the binary predicate 'rule' in Figure 9 refers to the database (non-rewritten) rules. The predicate 'evaluate' represents access to the facts. Given a query Q, 'evaluate(Q)' expresses the evaluation of Q on the already generated facts.

---

1. $fact(Q) \leftarrow query(Q) \wedge rule(Q, B) \wedge evaluate(B)$
2. $query(B) \leftarrow query(Q) \wedge rule(Q, B)$
3. $query(Q_1) \leftarrow query(Q_1 \wedge Q_2)$
4. $query(Q_2) \leftarrow query(Q_1 \wedge Q_2) \wedge evaluate(Q_1)$

Fig. 9

The bottom-up evaluation of Rule 1 on a database DB produces an answer $Q\sigma$ to a query Q if $Q\sigma \in T(DB)$, i.e., if $Q\sigma$ is an immediate consequence of DB. Answers that are not immediate consequences are obtained through Rules 2, 3 and 4 as follows. The bottom-up processing of Rule 2 generates a new subgoal 'query($B\sigma$)' in presence of a goal 'query(Q)' if Q unifies by σ with the head H of a database rule $H \leftarrow B$. Processing bottom-up Rules 3 and 4 evaluate conjunctive subgoals from left to right. Figure 10 illustrates how processing the meta-rules of Figure 9 bottom-up performs top-down reasoning from the goal 'query(t(a, x))' on the database of Figure 8.

The sets Δ_i of Figure 10 denote the difference sets

$$T\uparrow^i(DB) \setminus T\uparrow^{i-1}(DB)$$

generated by applying the semi-naive method on the meta-rules of Figure 9 and the following meta-facts:

query($t(a, x_1)$)
rule($t(x, y)$, $p(x, y)$) (i)
rule($t(x, z)$, $p(x, y) \wedge t(y, z)$) (ii)
$p(c_1, c_2)$ for each arc $c_1 \rightarrow c_2$ of Figure 8

Δ_1:	fact(t(a, b))	*from rules*	1-(i)
	fact(t(a, d))		1-(i)
	query($p(a, y) \wedge t(y, z)$)		2-(ii)
Δ_2:	query(p(a, y))		3
	query(t(b, z))		4
	query(t(d, z))		4
Δ_3:	fact(t(b, c))		1-(i)
	fact(t(d, c))		1-(i)
	fact(t(d, e))		1-(i)
Δ_4:	fact(t(a, c))		1-(ii)
	fact(t(a, e))		1-(ii)

Fig. 10

Facts in Δ_{i+1} follow from facts in Δ_i. In particular, 'fact(t(a, c))' in Δ_4 is derivable by Rules 1 and (ii) since fact(t(b, c)) $\in \Delta_3$. Indeed, this implies that

evaluate($p(a, b) \wedge t(b, c)$)

is true in $T\uparrow^3(DB)$. Since $T\uparrow^5(DB) \subseteq T\uparrow^4(DB)$, $\Delta_5 = \varnothing$ and the generation of facts and queries stops.

The meta-interpreter of Figure 9 generates non-first-normal-form tuples such as 'query(t(b, z))'. Normalized tuples can be obtained by specializing this meta-interpreter with respect to the rules of the considered database, i.e., by applying partial evaluation [SES 87] on the rules of Figure 9. Consider for example Rule 1. If the database under consideration is the one of Figure 8, the expression 'rule(Q, B)' denotes either rules (i) or (ii). Therefore, the relevant instances of Rule 1 are:

$$\text{fact}(t(x, y)) \leftarrow \text{query}(t(x, y)) \wedge \text{rule}(t(x, y), p(x, y)) \wedge \text{evaluate}(p(x, y))$$

$$\text{fact}(t(x, z)) \leftarrow \text{query}(t(x, z)) \wedge \text{rule}(t(x, z), p(x, y) \wedge t(y, z)) \wedge \text{evaluate}(p(x, y) \wedge t(y, z))$$

These instances can be simplified into:

$$t(x, y) \leftarrow \text{query}(t(x, y)) \wedge p(x, y)$$
$$t(x, z) \leftarrow \text{query}(t(x, z)) \wedge p(x, y) \wedge t(y, z)$$

Specializing the relation 'query' with respect to t and p normalizes tuples like 'query(t(x, y))' and 'query(p(x, y))' into 'query-t(x, y)' and 'query-p(x, y)', respectively. The specialized query predicates are called 'problem' in the Alexander method, and 'magic' in the Magic Set method.

Finally, the AMS methods avoid the generation of non-ground tuples such as 'query-t(d, z)' by pre-encoding the variables appearing in the heads of the specialized versions of Rules 3 and 4 by means of so-called adornments. For example, a non-ground head magic-t(d, z) is expressed as magic-t^{bf}(d) where the adornment *bf* means that the constant 'd' is the first attribute. It is worth noting that although a reordering of the body literals in the adorned rules is often desirable for improving efficiency, it is not necessary for achieving soundness, completeness, or termination of the AMS methods.

The above-defined specialization of the meta-rules of Figure 6 results in the rewriting algorithm of the Magic Set method [B* 86]. The more efficient rewriting of the Alexander method [R* 86] – re-discovered with the Supplementary Magic Set method [BR 87] – requires to refine the partial evaluation – see, e.g., [BRY 89].

6. Advantages of Upside-down Meta-interpretation

How efficient Satchmo, the AMS methods might be, it seems that direct implementations of their reasoning principles – i.e., bottom-up for Satchmo, top-down for the AMS methods – should be even more efficient. In this section, we refute this intuition. We show that Satchmo and the AMS methods efficiently rely on the search strategy of the underlined inference engine and give rise to combine bottom-up and top-down reasoning in a same process. Finally, we investigate other advantages of the AMS approach to top-down reasoning.

6.1. Inheriting Search Strategies

Model building à la Satchmo extends bottom-up reasoning with a processing of disjunctions by case analysis. Due to the simplicity of bottom-up reasoning, procedures like the naive or semi-naive methods are – up to unification – easily implemented in any programming language. Since Prolog is non-deterministic and provides us with a unification procedure, it is especially convenient for this task. The case analysis, however, is much more complex and, at first, seems rather difficult to implement efficiently in Prolog.

There are two basical approaches to case analysis: The various cases can either be considered at the same time, or the one after the other. The first approach is a breadth-first search of the various possibilities. The second corresponds to a depth-first search. It is retained by Satchmo. The deapth-first strategy requires to set up pointers to the yet non-explored cases, in order to allow searching them later.

Implementers of top-down reasoning systems are faced to a similar alternative. The various rules whose heads match the goal under consideration can, on the one hand, be considered at once. On the other hand, they can be considered

the one after the other. The first strategy performs a breadth-first expansion of proof trees, the second proceeds depth-first. The depth-first strategy is usually retained for top-down reasoning system, e.g., for Prolog interpreters. Indeed, the depth-first strategy gives rise to use the stack data structure for storing the subgoals, and this data structure can be very efficiently managed in main memory. For conventional automated reasoning applications, storing subgoals in main memory is not a serious restriction. In particular, it is a reasonable choice in theorem proving.

The programs of Satchmo are based on the analogy between depth-first case analysis and depth-first top-down reasoning. With the procedure 'component' Satchmo searches the various cases – i.e., the various components of disjunctions – by relying on the Prolog ability to search for clauses whose heads unify with a goal. The procedure 'component' does not explicitly set pointers to the next component of a disjunction. This is done by the Prolog interpreter itself. By the procedure 'assume', components of disjunctions are represented as Prolog facts. The backtracking facility of Prolog restores the context while moving from one case to the next. Thus, Satchmo uses Prolog like a programming language dedicated to specifying case analyses.

The shortness of Satchmo's programs results from this unconventional use of Prolog: The complex part of model building – i.e., case analysis – is not really implemented, but merely inherited from the Prolog interpreter. The efficiency of Satchmo comes to a large extend from the fact that Prolog interpreters are especially designed for an efficient search of clauses, for an efficient backtracking. Implementing a model building method in a conventional manner would force one to implement data structures and search mechanisms similar to those of Prolog. Considered from this angle, Satchmo is a more direct implementation!

Like Satchmo, the AMS methods inherit the search strategy of the considered inference engine. In contrast with implementations based on SLD-Resolution, they do not explicitly implement the component of breadth-first search which ensures termination in presence of recursive rules. They inherit it from the semi-naive method. Therefore, the AMS methods should be viewed as more direct implementations than the procedures based on SLD-Resolution.

6.2. Combining Bottom-up and Top-down Reasoning

Another important feature of upside-down meta-interpretation – à la Satchmo as well as à la Alexander and Magic Set – is to permit one combining bottom-up and top-down reasoning. Consider the rules of Figure 7. Most of them are Horn rules. Therefore, it is possible to write them as Prolog rules instead of relying on the 'rule' meta-predicate. This approach often dramatically improves the efficiency of Satchmo [MB 88].

This is in particular the case if the example considered above is augmented with some facts for s. If the rule 't(x) ← s(x)' is processed bottom-up, these additional facts would induce new t facts. These facts are not needed for proving inconsistency since the original example is itself inconsistent. If the rule 't(x) ← s(x)' is evaluated top-down, then only the query t(a) is posed during model building. Less deductions are performed.

Similar examples can be given for the AMS methods, for which processing certain rules bottom-up – i.e., keeping them unchanged instead of rewriting them – is preferable. An upside-down meta-interpreter always gives rise to process some rules bottom-up, others top-down. One reasoning principle is offered by the inference engine and the other by the meta-rules.

6.3. Top-down Reasoning with Relational Data Structures

Relying on the semi-naive method for implementing top-down reasoning has other important advantages. First, this yields a normalized data structure more suited to the database context than the evaluation stack of Prolog. Storing queries as facts in relations helps searching for redundant queries. Relying on the hierarchical stack data struc-

ture would make this search more expensive. Moreover, using a relational data structure makes possible to rely on database facilities for storing queries on secondary memory, if necessary. Finally, relying on the semi-naive method is prefereable from an engineering viewpoint. In many cases, e.g., for completely computating a transitive closure, bottom-up processing is more efficient than top-down reasoning. Therefore, database systems need a semi-naive evaluator. As the AMS methods rely on such an evaluator they give rise to rather simple implementations. In contrast, other implementations of the same top-down reasoning principle make use of another, slightly different evaluator [LV 89]. We think that they are more complex. In addition, these implementations induce some redundancies in the database system since two versions of a same evaluator are needed.

7. Conclusion

In this article, we have analyzed methods that were proposed for enhancing database systems with automated reasoning. We have considered on the one hand the theorem prover Satchmo, on the other hand the Alexander and Magic Set methods. Satchmo was developed for verifying the consistency of database integrity constraints. The Alexander and the Magic Set methods were proposed for evaluating queries on recursive databases. These methods introduce new ideas, some of them inspired from databases. In particular, Satchmo makes use of safety or range-restriction and treats negation by failure. As opposed to conventional automated reasoning procedures, the Alexander and Magic Set (AMS) methods rely on relational, normalized data structures.

However, the most original characteristic of Satchmo and of the AMS methods is the use of a technique we called upside-down meta-interpretation. Satchmo implements a bottom-up reasoning theorem prover in the top-down language Prolog. The AMS methods can be formalized as a meta-interpreters that rely on the bottom-up semi-naive method for implementing top-down reasoning. This unconventional use of meta-interpretation has not deserved much attention in the past. The article [GCS 88], which describes an approach similar to that of the AMS methods, seems to be a noticeable exception. We have investigated upside-down meta-interpretation by refering to Satchmo on the one hand, and to the AMS methods on the other hand.

Upside-down meta-interpretation gives rise not to re-implement the desired search strategy – depth-first or breadth-first – but to inherit it from the considered inference engine – Prolog for Satchmo, the semi-naive procedure for the AMS methods. It also permits one to combine the bottom-up and top-down reasoning principle during a same deductive process. This approach, which was shown very successful with Satchmo, seems to be promising for optimizing query evaluation in databases. An open question is the definition of strategies for deciding when to reason bottom-up, and when to proceed top-down.

With the AMS methods, upside-down meta-interpretation benefits from partial evaluation. However, this technique does not seem to be relevant for Satchmo. It would be interesting to know under which conditions it is beneficial to specialize by partial evaluation an upside-down meta-interpreter.

Another open question is the use of upside-down meta-interpretation in other contexts than those considered in this article. First experiments let us think that this technique can also be applied for other automated reasoning tasks, yielding simple and efficient implementations.

Finally, the approach of Satchmo for handling rules with disjunctive heads could be applied for querying disjunctive databases. Although the Prolog implementation of Satchmo does not seem to be applicable in such a context, the experience we made with Satchmo could be of interest for large databases as well.

Acknowledgements

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.

I am indebted to Alexandre Lefebvre, Catriel Beeri, Rainer Manthey, and my colleagues in "Compulog" for helpful discussions.

References

[B* 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.

[BDM 88] F. Bry, H. Decker, and R. Manthey. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *Proc. 1st Int. Conf. Extending Database Technology (EDBT).* 1988.

[BM 86] F. Bry and R. Manthey. Checking Consistency of Database Constraints: A Logical Basis. In *Proc. 12th Int. Conf. on Very Large Data Bases (VLDB).* 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).* 1987.

[BRY 89] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases (DOOD).* 1989.

[DIE 87] S.W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proc. Symp. on Logic Programming (SLP).* 1987.

[FRE 87] J.C. Freytag. A Rule-Based View of Query Optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD).* 1987.

[G* 85] J. Gabriel, T. Lindholm, E.L. Lusk, and R.A. Overbeek. *A Tutorial on the Warren Abstract Machine.* Technical Report ANL-84-84, Argonne National Laboratory, 1985.

[GCS 88] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing* 6:159-186, 1988.

[GDW 87] G. Graefe and D.J. DeWitt. The EXODUS Optimizer Generator. In *Proc. ACM SIGMOD Conf. on the Management of Data (SIGMOD).* 1987.

[KSS 87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB).* 1987.

[LOV 68] D.W. Loveland. A Linear Format for Resolution. In *Proc. IRIA Symp. on Automatic Demonstration.* 1968.

[LUC 68] D. Luckham. Refinements Theorems in Resolution Theory. In *Proc. IRIA Symp. on Automatic Demonstration.* 1968.

[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).* 1989.

[MB 88] R. Manthey and F. Bry. SATCHMO: A Theorem Prover Implemented in Prolog. In *Proc. 9th Int. Conf. on Automated Deduction (CADE).* 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).* 1987.

[R* 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.

[SES 87] P. Sestoft and H. Søndergaard. A Bibliography on Partial Evaluation. *SIGPLAN Notices* 23(2), 1987.

[STS 86] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Mass., 1986.

[TAR 55] A. Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics* 5, 1955.

[TS 86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proc. 3rd Int. Conf. on Logic Programming.* 1986.

[VIE 87] L. Vieille. A Database-complete Proof Procedure Based on SLD-resolution. In *Proc. 4th Int. Conf. on Logic Programming.* 1987.