DEDUCTIVE INFORMATION RETRIEVAL ON VIRTUAL RELATIONAL DATA BASES

M.H. van Emden

Research Report CS-76-42

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

August 1976

1. Synopsis

The synopsis is divided into several subsections, each introducing a topic of the present paper.

1.1 Some practical problems in connection with data bases

It is the ambition for a central data base of an organization to be the repository of information required for all aspects of operation. Many of these aspects are intimately related. A data base capable only of reproducing explicitly present information must be highly redundant. If for no other reason, deduction is valuable in an information retrieval system because it allows elimination, or at least control, of redundant information.

Consider the analogy of a computer program requiring values of the logarithmic function. Such a program need not incorporate a bulky table of logarithms, but it uses instead a compact subroutine which computes the required values. Likewise, data bases typically contain in tabular form large numbers of explicit facts which could be deduced from a compactly stored rule.

Thus, a University might decree that all graduate courses have a course number of at least 500, thus making it redundant to store for courses thus numbered the fact that they are graduate courses. The redundancy can be avoided if the data base can store besides facts a rule such as this and if the redundant facts thus suppressed can be deduced on demand. As another example, all first-year engineering students may have to take a course named Math 129; the deductive use of this rule would appreciably shorten in the data base the list of students taking Math 129.

There are problems also in connection with the query language. For example, the difficulties of the typical casual user with Codd's query languages have prompted Zloof's "query-by-example" as a method of interrogating a data base. Another example is the fact that the data retrieved by queries are often not required for inspection by humans, but serve as input for programs, for example for statistical processing. Such usage places high demands on the interface between query language and program language.

1.2 A logical model for information representation and retrieval

The various aspects of a data base system are represented in terms of the clausal form of first-order predicate logic and a resolution refutation procedure. In this respect the paper is self-contained; it includes all material required for a full understanding by those not familiar with resolution theorem-proving. The main points of the representation are as follows.

Virtual relational data base:

A set DB of clauses, each with one positive literal.

Query:

A clause Q containing no positive literal.

Affirmative response:

Occurs if $DB \cup \{Q\}$ is found to be inconsistent by the refutation procedure.

Negative response:

Occurs if non-existence of refutation from $DB \cup \{Q\}$ can be shown by the refutation procedure.

Answer to a query:

In case of an affirmative response the refutation procedure determines a substitution θ such that $DB' \cup \{Q\theta\}$ is inconsistent, where DB' is a set of instances of clauses in DB. From $Q\theta$ the answer to Q may be extracted; the answer is logically implied by DB.

Relation retrieved:

Each different refutation of the same query Q contributes one tuple to the relation retrieved by Q.

1.3 The virtual relational data base

In the relational view of a data base the data items are organized as sets of similar **tuples** of items. Each such set is a relation according to the usual mathematical definition. In the existing concept of a relational data base new relations are composed by queries, but only by queries. In the present model such composition is defined by "rules" also stored in the data base. A rule defines a relation which is not present as a set of tuples, yet any of its tuples can be deduced on demand by a query. Hence a *virtual* relational data base. Some queries are answered by look-up only: the required tuple is stored. Other queries may require a lengthy deduction, using a rule many times before a stored tuple can be used. In a typical information retrieval application look-up will be the rule, and rule application the exception. But it is not necessary to use a virtual relational data base this way.

1.4 A relational model for both data and computation

A function can be viewed as a set of (argument, value) tuples. Because the set is often very large (or not even finite) it can only exist as a data base if the data base is virtual. For example, the data base (6.2) stores a rule recursively defining the factorial function and the tuple (0,1) for terminating the recursion. Deductive information retrieval computes the factorial function for arbitrary values in much the same way as other models of computation do. In our model information retrieval and computation differ in degree rather than in kind. If answering a query involves mostly look-up, then it is retrieval in the usual sense. If it involves mostly deduction, then it is computation in the usual sense. Deductive information retrieval on a virtual relational data base is therefore a model of computation. This model originated with Kowalski, who called it logic programming. It has been shown [12] to generalize other computation models in an interesting way, and to have a simple semantics related to the usual fixpoint semantics [6]. A practically useful program language (PROLOG) is based on it. In our approach the query language is essentially this program language. In this way the problem of the interface between query languages and program language is solved in a particularly elegant way: the languages are the same.

1.5 Minimal model semantics

It is useful to distinguish "first-order queries" from "second-order queries". In order to answer a first-order query, a composition of relations of the data base is computed. Examples are the compositions called projection or join in Codd's relation algebra. Answers to such queries are true in *all* models.

Certain other queries are expressed by an inclusion among relations retrieved by first-order queries, and are therefore called second-order queries. Examples are those queries expressed by the "division" operation in Codd's relation algebra. Answers to these queries are not true in all models, hence are not logical implications, but they are true in the *minimal model*. We show that the minimality of the model here plays the same role as the minimality of the fixpoints in Scott's approach to the theory of computation.

2. Related work

A more powerful system for deductive information retrieval has been designed by R. Reiter [19]. Internally it uses the clausal form of first-order predicate logic, but the query language is a non-clausal logic geared to translation from natural language input. Indefinite facts can be stored in the data base and can be provided as answer.

Kowalski himself applies logic programming to information retrieval [14]. His research involves the design of a data base concerning all aspects of the operation of his university department. The emphasis in Kowalski's project is on flexibility in data base structure. For instance, it is easy to add or delete a domain of a relation. The query language is clausal form; Kowalski shows that a few minor syntactical enhancements suffice to make queries so easy to understand that natural language is hardly worth having.

J. Minker [8] was early to make use of the basic fact that in first-order predicate logic one cannot help adopting a relational view of data and that resolution theorem-proving can provide a powerful query language for a relational data base.

Our work is distinct from the above approaches in being based on the premiss that the design of the refutation prodedure used as language interpreter in PROLOG can be adapted to efficient information retrieval by incorporating the indexing schemes and search algorithms used in implementing existing relational data bases. The objective is to obtain a useful query language which is a powerful program language in its own right.

The most important previous work is Green's on the application of resolution theorem-proving to question-answering [9] and Kowalski's on the application of logic to programming in a high-level language.

3. Syntax and informal semantics

We will use the clausal form of first-order predicate logic as the abstract representation of a virtual relational data base. Terms of logic will represent objects; the clauses will represent the answers that constitute the data base.

A syntax for the clausal form of first-order predicate logic can be specified as follows. Let a non-empty sequence of letters or digits be called an *identifier*. A *constant* is an identifier and should be read as denoting an unstructured object. For example

Math
129
Glotz

A *term* is a simple term or a composite term. A *simple term* is a constant or a variable. A *variable* is an identifier preceded by an asterisk. A *composite term* is $f(t_1,\ldots,t_n), n \geq 1$, where f is a function symbol and $t_1,\ldots,t_n$ are terms, the *arguments* of the composite term. A *function symbol* is an identifier or one of the *operator symbols*

. : ; ! @ &

A *substitution* is the replacement of all occurrences of a variable in an expression (a term, or other expression described later) by a term. If e_2 is the result of applying a substitution to an expression e_1 then e_2 is called an *instance* of e_1.

A composite term is to be read as denoting a structured object; the function symbol indicates how the components of the structured object are assembled. A term containing variables is to be read as an incompletely specified object: it is not specified which of its variable-free instances it denotes.

For example

$$:(.(C,.(Y,K)),Cheng)$$

has as components the composite term $.(C,.(Y,K))$ and the simple term Cheng.

For a two-place function symbol which is an operator, infix notation is permitted, so that we may write

$$C.Y.K: \ Cheng$$

instead of

$$:(.(C,.(Y,K)),Cheng)$$

provided it has been made clear somehow that . and : are two-place function symbols, that . has higher priority than :, and that . associates from right to left, i.e. that, for example, C.Y.K stands for C.(Y.K) rather than (C.Y).K.

A *sentence* is a set of clauses.

A *clause* is a set $\{L_1,\ldots,L_n\}$, $n \geq 0$, of literals written as

$$L_1 \ \ldots \ L_n \qquad \text{for } n > 0$$

and as

$$\square \qquad \text{for } n = 0$$

The empty clause is often called the *null clause*.

A *literal* is $+A$ (and then it is a *positive literal*) or $-A$ (and then it is a *negative literal*), where A is an atom. An *atom* is $P(t_1,\ldots,t_k)$, $k \geq 0$, where P is a k-place predicate symbol and $t_1,\ldots,t_k$ are terms, the *arguments* of the atom. A *predicate symbol* is an identifier.

Table 3.1 shows a sentence with several kinds of clauses, all of which are special cases of

$$+A_1 \ \ldots \ +A_n \ - B_1 \ \ldots \ -B_m \qquad \ldots(3.1)$$

In the case, where $n \geq 1$ and $m \geq 0$, the clause is to be read as

for all $x_1,\ldots,x_k$, A_1 or...or A_n

if there exist $y_1,\ldots,y_j$ such that B_1 and...and B_m

where $y_1,\ldots,y_j$ are the variables of $B_1,\ldots,B_m$ and $x_1,\ldots x_k$ are the remaining variables.

In this paper a sentence is often viewed as a data base, and the clauses in it as answers to potential queries, or as rules for answering such queries. A clause such as the above plays the role of an

indefinite, conditional rule for answering. Indefinite because it does not say which, if any, of $A_1,\ldots,A_n$ is false. Conditional because A_1 of...or A_n can only be used as an answer if the query

do there exist $y_1,\ldots,y_j$ such that $B_1,\ldots,B_m$?

is affirmatively answered. It is a rule for answering rather than an answer in case the clause contains variables.

A *definite* clause will mean a clause with one positive literal and zero or more negative literals, that is, n must be 1. A *definite* sentence is one where all clauses are definite. For reasons to be explained later, a data base has to be a definite sentence.

Example Clause (1) in Table 3.1 is to be read as

for all x, x takes Math 129
  if x is first year and in the engineering program

Clause (8) in Table 3.1 is to be read as

for all x, x is a graduate course
  if there exists a y such that y is the
  course number of x and y is greater than 499

That is, we have for answering certain queries the rule: Graduate courses have numbers greater than 499.

If in the clause (3.1) we have $n = 1$ and $m = 0$, then we have an unconditional answer. If such a clause has no variable in it, then it unconditionally asserts one specific fact. The subset in a sentence of all variable-free clauses containing one positive literal and no negative literal and having the same, say k-place, predicate symbol, is called an *array*. The reason for this is that the set of k-tuples of arguments specifies a relation in the same way as it is done by an "array" in the sense of Codd [2].

The usual relational data base consists of arrays only. The presence of rules provides the possibility of answers which are not explicitly present, hence makes the data base virtual. We owe the idea of having rules coexist with arrays to Reiter [19], where the collection of arrays is called the extensional data base and the rules are called the intensional data base. Reiter's system is not restricted to definite rules or facts.

Table 3.2 shows an example of an array in set notation.

Table 3.3 shows the same array in a less redundant notation and this is also used in the arrays of table 3.4.

The clauses containing one positive literal and no negative literals are more general than the tuples of a conventional relational data base because they may contain variables. For example, the term *x:*y in (6) and (7) of Table 3.1 is an incompletely specified object, the general form of the name of a person; (6) states that what comes before the colon is the sequence of initials of the name, like J.F in J.F:Glotz; (7) states that what comes after the colon is the last name of the name, like Glotz in J.F:Glotz.

RULES *) =

```
(1)  {+Takes(*x,Math!129) - Year(*x,1) - Program(*x,Engineering)
(2)  ,+Year(*x,*z) - Student(*x,*y,*z1) - Minus(1977,*z1,*z)
(3)  ,+Program(*x,*y) - Student(*x,*y,*z)
(4)  ,+Courseprefix(*x!*y,*x)
(5)  ,+Coursenumber(*x!*y,*y)
(6)  ,+Initials(*x:*y,*x)
(7)  ,+Lastname(*x:*y,*y)
(8)  ,+Graduatecourse(*x) - Course number(*x,*y)
                          - Greaterthan(*y,499)
(9)  ,+Conflict1(*x1,*x2) - Scheduled(*x1,*y,*z)
                          - Scheduled(*x2,*y,*z)
                          - Different(*x1,*x2)
     }
```

Table 3.1

*) We gratefully acknowledge Kowalski's departmental data base as the source of inspiration for the present example.

```
{ +Takes(M     :Adiri, Math!129)
 ,+Takes(C.Y.K:Cheng, Math!225)
 ,+Takes(T.L   :Cook , Math!129)
 ,+Takes(T.L   :Cook , Math!225)
 }
```

Table 3.2

Takes		
	M :Adiri	Math!129
	C.Y.K:Cheng	Math!225
	T.L :Cook	Math!129
	T.L :Cook	Math!225

Table 3.3

Teaches		
	J.F:Glotz	Math!129
	J.F:Glotz	Math!225

Student			
	M :Adiri	Biology	1974
	N.A :Buczek	Recreation	1976
	C.Y.K:Cheng	Physics	1976
	T.L :Cook	Engineering	1975
	G.C :Giusti	Engineering	1976
	A :Hammer	Child Care	1973
	K.L :Mensink	Kinesiology	1973

Scheduled			
	Math!129	Phy@3009	2:30
	Math!301	Phy@3009	2:30

<u>Table 3.4</u>

4. Semantics for clausal form

The semantics of first-order predicate logic determines whether a sentence is true. In a later section we will use the clausal form of first-order predicate logic to represent the *facts* and the *rules* of a virtual relational data base, and also the queries that can be submitted to it. The semantics defined in this section, which is the usual one in substance (though the elegantly simple form is due to Kowalski [6]), will be used to determine which are truthful answers to queries.

The set of all variable-free terms that contain only constants or function symbols occurring in a sentence S, is called the *universe of discourse* of S. The set of all atoms that contain only predicate symbols of S and terms of the universe of discourse of S, is called the *universe of atoms* of S. An *interpretation* for S is a subset of its universe of atoms. Now it shall be determined whether S is true in an interpretation I.

A sentence is *true in I* if no clause is not true in I.

{Hence a sentence should be read as the conjunction of its clauses}

A clause is *true in I* if all its variable-free instances are true in I.

A clause C' is a *variable-free instance* of C if C' is obtained by replacing variables in C by terms from the universe of discourse. Different occurrences of the same variable are replaced by the same term.

A variable-free clause is *true in I* if at least one of its literals is true in I.

{Hence a clause is to be read as the universally quantified disjunction of its literals; the null clause is true in no interpretation}

A variable-free literal +A is *true in I* if $A \in I$.

A variable-free literal −A is *true in I* if $A \notin I$.

It is not always most helpful to read a clause as a disjunction, although that is most obviously correct. Let us verify that it is also correct to read a clause

$$+A_1 \ldots +A_n \; -B_1 \ldots -B_m, \quad n > 0, \qquad \ldots(4.1)$$

(where $y_1,\ldots,y_j$ $(j \geq 0)$ are the variables in $B_1,\ldots,B_m$ and where $x_1,\ldots,x_k$ $(k \geq 0)$ are the remaining variables) as

$$\text{for all } x_1,\ldots,x_k,\ A_1 \text{ or } \ldots \text{ or } A_n \text{ if}$$
$$\text{there exist } y_1,\ldots,y_j \text{ such that } B_1 \text{ and } \ldots \text{ and } B_m \qquad \ldots(4.2)$$

Suppose that (4.1) is not true in an interpretation I. Then, according to the formal semantics there exists a variable-free instance, say $(+A_1 \ldots +A_n -B_1 \ldots -B_m)\ \theta$, of (4.1), say with terms $x'_1,\ldots,x'_k$, $y'_1,\ldots,y'_j$ as values for $x_1,\ldots x_k$, $y_1,\ldots,y_j$, which is not true in I. Hence $A_i\ \theta \notin$ I for $i = 1,\ldots,n$ and $B_i\theta \in$ I for $i = 1,\ldots,m$. Therefore, there exist $y_1,\ldots,y_j$ (namely $y'_1,\ldots,y'_j$) such that B_1 and ... and B_m and not (A_1 or ... or A_n) for some $x_1,\ldots,x_k$ (namely $x'_1,\ldots,x'_k$); this contradicts (4.2). The converse is left to the reader.

For n = 0, the clause (4.1) is true in I if at least one of $B_1,\ldots,B_m$ is not true in I. Hence the clause is to be read as

$$\text{for all } y_1,\ldots,y_j,\ \text{not } (B_1 \text{ and } \ldots \text{ and } B_m)$$

A sentence is *consistent* if it is true in at least one interpretation. Such an interpretation is called a *model* for S. A sentence S_1 is a *logical implication* of a sentence S_2:

$$S_2 \models S_1$$

if S_1 is true in all models of S_2. Let S'_1 be a negation of S_1 in the sense that S_1 and S'_1 have the same universe of atoms and S'_1 is not true in every interpretation where S_1 is true. Then $S_2 \models S_1$ iff $S_2 \cup S'_1$ inconsistent, provided S_2 is consistent.

5. Logic representation of an information retrieval system

As soon as it is possible for an information retrieval system to produce answers not explicitly present in the data base, it is important to know exactly in what sense such answers are warranted by the contents of the data base. As a first approximation we are interested in answers which are logical implications of the data base when the latter is regarded as a sentence of first-order predicate logic. We have already defined precisely, as a semantic notion, logical implication. The problem remains how to compute just those logical implications which can be regarded as an answer to a query. For such computations we need a proof procedure.

The proof procedure will be the logical model of the retrieval component of the information retrieval system. In the clausal form of first-order predicate logic the question of logical implication of S_1 by S_2 is expressed as the inconsistency of $S_2 \cup S_1'$, where S_1' is a negation of S_1. A procedure for demonstrating inconsistency is called a refutation procedure. We consider refutation procedures which refute S by constructing a sequence $S_0, \ldots, S_n$ of sentences such that $S_o = S$, $M(S_i) = M(S_{i-1})$, for $i = 1, \ldots, n$, and $\square \in S_n$. Here $M(x)$ denotes the set of models of x. Each S_i has been obtained from S_{i-1} by applying a *rule of inference* to S_i.

The rule of inference obtains S_i by replacing in S_{i-1} a clause

$$-A_1 \ldots -A_i \ldots -A_n,\ n \geq 1,\ 1 \leq i \leq n,$$

which is one *parent* in the inference, by

$$(-A_1 \ldots -A_{i-1}\ -B_1 \ldots -B_m\ -A_{i+1} \ldots -A_n)\theta$$

whenever S_{i-1} also contains a clause

$$+A\ -B_1 \ldots -B_m$$

which is the other parent in the inference, such that $A\theta = A_i\theta$, where θ is the most general unifier of A and A_i. A *unifier* θ of atoms k_1 and k_2 is a substitution of terms for variables such that $k_1\theta = k_2\theta$; the unifier is also a *most general unifier* if for any unifier θ' of k_1 and k_2, $k_1\theta' = k_2\theta'$ is an instance of $k_1\theta = k_2\theta$. J.A. Robinson showed [20] that whenever a unifier exists, a most general unifier exists, and that this unifier is essentially unique. He also found an algorithm that either constructs the most general unifier, or shows that none exists. Paterson and Wegman [18] have published a unification algorithm that requires an amount of time and space which is linear in the length of the input expressions.

A_i is the unique result of applying the selection rule of the refutation procedure to the parent

$$-A_1 \ldots -A_n$$

The selection rule severely restricts the number of different applications of the inference rule.

The rule of inference is a refinement of J.A. Robinson's resolution principle [20], which was developed by Kowalski and Kuehner [15], Kuehner [16], and Kowalski [13].

The fact that $M(S_{i-1}) = M(S_i)$ expresses the "soundness" of the rule of inference: the inferred sentence S_i is true in exactly those situations where S_{i-1} is. It is in this sense that an information retrieval system, as described here, will not "tell a lie".

The logic representation of the various aspects of an information retrieval system can be summarized as follows.

Virtual relational data base:

A set DB of clauses ("answers" of various kinds), each with one positive literal.

Query:

A clause $Q = -A_1 \ldots -A_n$ $(n \geq 0)$ containing no positive literal.

Affirmative response:

Occurs if $DB \cup \{Q\}$ is found to be inconsistent by the refutation procedure.

Negative response:

Occurs if nonexistence of a refutation from $DB \cup \{Q\}$ can be shown by the refutation procedure.

Answer to a query:

In case of an affirmative response the refutation procedure determines a substitution θ such that $DB' \cup \{Q\theta\}$ is inconsistent, where DB' is a set of instances of clauses in DB. The inconsistency of $DB' \cup \{Q\theta\}$ is equivalent to

DB $\models_{\overline{T}} \{+A_i\theta\}$ for i = 1,...,n. The sentence $\{+A_1\theta,\ldots,+A_n\theta\}$ is an *answer to query Q*. The clauses $+A_1\theta,\ldots,+A_n\theta$ are answers that may or may not belong to the set DB. In the latter case such a clause has been deductively retrieved. Its presence in DB would be redundant.

Relation retrieved:

In case of an affirmative response to a query, with variables $x_1,\ldots,x_k$ $(k \geq 0)$, there may be more than one substition θ as described above. Each such θ determines a k-tuple of terms substituted for $x_1,\ldots,x_k$. The set of these tuples is the *relation retrieved by Q*.

Retrieval procedure:

Resolution refutation procedure for first-order predicate logic in clausal form.

Note that an $S_o = DB \cup \{Q\}$ contains one clause, namely Q, without a positive literal and all its other clauses (those of DB) contain exactly one positive literal. With such an S_o all sentences S_i of a refutation have the same property, in fact, $S_i = DB \cup \{Q_i\}$ with Q_i containing no positive literal. Apparently the refutation procedure acts as an information retrieval procedure by successively transforming the original query Q to queries $Q_1,\ldots,Q_i,\ldots$ until the empty query is obtained.

Example 5.1

Let DB = TAKES ∪ TEACHES ∪ SCHEDULED ∪ STUDENT ∪ RULES
as given in Tables 3.1,...,3.4.

Let Q_o = -Takes(*x,Math!129).

The rule of inference will use as parents Q_o and

+Takes(M:Adiri,Math!129) ∈ TAKES

and produce $Q_1 = \square$, the empty clause, in this case the empty query. The unifier θ substitutes M:Adiri for *x. Thus, there is an *affirmative response*, and the *answer* is

+Takes(M:Adiri,Math!129)

exactly a clause of the data base. In order to obtain the *relation retrieved*, the refutation procedure constructs all possible refutations. In a similar way the answer

+Takes(T.L:Cook,Math!129)

is produced, another clause from the data base. In an attempt to find yet another refutation, the rule of inference is now applied to Q_o and the conditional rule

```
+Takes(*x,Math!129) - Year(*x,1)
                    - Program(*x,Engineering)
```

which is a clause in RULES (see Table 3.1) saying that all first-year engineering students take Math 129. The original query Q_o is now replaced by

Q_1 = -Year(*x,1)-Program(*x,Engineering)

Starting from here the refutation procedure produces all first year engineering students. Let us suppose for convenience that the selection rule always selects the leftmost literal in a query. This part of the query is tackled by using as parent

+Year(*x,*z)-Student(*x,*y,*z')-Minus(1977,*z,*z')

a conditional rule, which refers queries about "Year" to the STUDENT array and to an imaginary array MINUS, supposed to contain all triples n_1,n_2,n_3 of integers such that $n_1-n_2 = n_3$. The last column of the student array gives the year of first registration. The second argument of "Year" tells that the current year (1977 according to the rule) is for a student with name x the z-th year of study. The array MINUS is imaginary because it is too large to be stored and queries to it can be answered by a simple computation.

Q_2 = -Student(*x,*y,*z')-Minus(1977,1,*z')-Program(*x,Engineering)

The first entry of the STUDENT array is now used as parent, resulting in

Q_3 = -Minus(1977,1,1974)-Program(M:Adiri,Engineering)

At this point no application of the rule of inference is possible. The refutation procedure backtracks to Q_2, the last point where there was a choice of parent. It finds another triple from the STUDENT array, and invokes MINUS to check whether this is a first-year student. Apparently, this refutation procedure *generates* successive answers to the first part of the query, and then *checks* each answer with the second part, but here it would be better to work the other way around.

In this example it would be better to execute

-Minus(1977,1,*z')

first, because there is a unique answer, which is then used to search STUDENT for all whose first year of registration is 1976. We assumed that the selection rule always selects the leftmost literal in a query. Although this has the advantage of simplicity, it is apparently not always optimal.

To summarize and complete the example, the query

Q = -Takes(*x,Math!129)

gives an affirmative response when

$DB = TAKES \cup TEACHES \cup SCHEDULED \cup STUDENT \cup RULES$

The query has several answers, namely

+Takes(M:Adiri,Math!129)
+Takes(T.L:Cook,Math!129)
+Takes(G.C:Giusti,Math!129)

one for each possible refutation of $DB \cup \{Q\}$. The first two answers are in the data base. The third answer has been deductively retrieved; its presence in the data base would be redundant.

The relation retrieved is the set of all substitution-tuples for the tuple of variables in the query, namely

{M:Adiri,T.L:Cook, G.C:Giusti}

Query-by-example [25] has a high degree of similarity with the queries defined here. This will be illustrated by means of the example queries Q, Q_1, and Q_2 discussed above. Note that there only Q originated with the user, which was then transformed by the refutation procedure successively to $Q_1, Q_2, \ldots$. But Q_1 and Q_2 could also have originated from a user, and can hence be usefully taken as example queries.

In query by example Q would be

Takes	Name	Course
	x	Math!129

Note the following differences. The columns have names, whereas in logic the arguments of an atom are identified only by their order. Variables are distinquished by underlining instead of by a preceding asterisk.

In query-by-example, Q_1 would be

Year		
	$\underline{x}$	1

Program		
	$\underline{x}$	Engineering

except that this time the required column names have been omitted.

In query-by-example Q_2 would be

Student			
	$\underline{x}$		$\underline{z'}$

Minus			
	1977	1	$\underline{z'}$

Program		
	$\underline{x}$	Engineering

Here we see an interesting feature of query-by-example: the variable second argument of Student need not be named. Independently, D. Warren of the University of Edinburgh has introduced the same feature into the program language PROLOG under the name "anonymous variable".

6. The relational model of computation

The relational model is not only useful for data, but also for computation. Consider the following example of a data base intended to answer queries about the values of the factorial function.

```
{+fact(0,1),+fact(1,1),+fact(2,2)                ... (6.1)
 ,+fact(3,6),+fact(4,24),...
 }
```

This is an actual relational data base without any opportunity for deductive information retrieval. Another possible data base is

```
{+fact(0,1)
 ,+fact(*x,*y) - minus(*x,1,*x')
               - fact(*x',*y')                   ... (6.2)
               - times(*y',*x,*y)
 }
```

As in example 5.1, the presence of an imaginary array MINUS is required. This time also, and in a similar way, an imaginary array TIMES. With the latter data base queries of the form

-fact(α,*y)

where α is a nonnegative integer, are, usually deductively, answered. The deduction mirrors exactly a computation according to the common recursive definition of factorial. It should be noted that refutation prodedures as described here will not answer most queries of the form -fact(*x,α) with the data base (6.2).

This example shows that the difference between computation and deductive information retrieval on a virtual relational data base is a difference in degree rather than in kind. In example 5.1 table look-up is the dominant operation, hence we have information retrieval. Here deduction is dominant, hence we have computation. The whole subject of logic programming [12,13,7] may be viewed as information retrieval on virtual relational data bases without any bias towards dominance of either look-up or deduction.

Michie's idea of a memo-function [17] should be mentioned in connection with this. Memo-functions are function definitions that contain an algorithmic part (our rules) and a data base part (our arrays) containing ("remembered") (argument,value) tuples that were computed previously. When calling a memo-function the user would not know whether he obtained a newly computed value or a retrieved value computed earlier. A memo-function autonomously deletes from and inserts into the data base in an attempt to optimize response time.

The relational model is not new to the theory of programming. Superficially, the object of computation is usually thought of as being a function. But nontermination in computation, and more generally in the execution of formal function definition, forces one to admit functions which are not everywhere defined. Also, indeterminacy in programs is incompatible with the single-valuedness of functions. Rather than to start speaking of partial, multivalued functions, it is preferable to realize that functions are total, single-valued binary relations, so that binary relations in their full generality are the appropriate model of what is computed. An important feature of Scott's method [22, 1] in the theory of computation is the explicit use of relations as sets of tuples, for example in the definition of the notion of approximation: a relation R_1 is an approximation to R_2 if $R_1 \subseteq R_2$, as sets of tuples.

7. PROLOG

As should be apparent from the previous section, a system for deductive information retrieval on virtual relational data bases modeled on the clausal form of first-order predicate logic will also be a system for (logic) programming. Apart from any applicability to information retrieval, logic programming, as for example embodied in PROLOG [4 ,21], is an important development in language design and programming methodology. Even with a fairly crude implementation good results have been obtained in computer understanding of natural language [4], robot plan formation [26], symbolic mathematical computation [11], and in compiler writing [4]. To this should be added Warren's results [24] in compiling PROLOG, which show that a language for logic programming is implementable at least as efficiently as LISP.

But all versions of PROLOG have an extreme bias towards deduction and away from retrieval. We believe this is not inherent in the logical model, but rather reflects the application that PROLOG implementers had in mind: symbolic computation. We believe that the indexing schemes and search algorithms required for efficient information retrieval are compatible with the overall design of existing PROLOG implementations and that there is a realistic prospect of a system for deductive information retrieval on a virtual relational data base that is at the same time a superior program language.

In the remainder of this section we will try to give the reader an idea of what kind of program language PROLOG is. Only superficial remarks will be made; for a proper understanding the reader is referred to an account of Kowalski's "procedural interpretation" of first-order predicat logic in clausal form [12,13]. There are two ways of looking at PROLOG: as an interpreter for a program language and as an implementation of a resolution refutation procedure.

Regarded as a program language, PROLOG occupies a position similar to LISP: what the lambda-calculus is for LISP, first-order predicate logic (in clausal form) is for PROLOG. There are two ways of praising a language: to describe the valuable features the language has and to list the questionable features that are lacking.

To start with the latter category: PROLOG has no goto statements, no assignment statements, indeed no statements of any kind, being a declarative rather than an imperative language. PROLOG does have procedure call by pattern matching. Both procedure declaration and procedure call are in several respects simpler and more general than, say, in Algol 60. This is appropriate since the lack of most of the usual features leaves the procedure mechanism as the programmer's only tool. The data structures are similar to Hoare's "recursive data structures" [10]. None of this would be of any use without PROLOG's library of system-defined procedures for input and output, arithmetic, runtime additions to or deletions from a program, programmer-defined deviations from the normal course of control, runtime interrogation of the state of the computation, and more.

The other way of looking at PROLOG is as a resolution refutation procedure as described in section 5. A surprisingly large part of the advanced language features of PROLOG arise in the course of the normal operation of the refutation procedure. This phenomenon has been discovered by Kowalski who called it "the procedural interpretation of first-order predicate logic". We will mention here only the following aspects of the procedural interpretation: unification is pattern-matching, resolution is procedure call, substitution is the mechanism for replacing formal by actual parameters.

The method followed in this paper echoes Kowalski's procedural interpretation. The method could well have been called "the data-base interpretation of first-order predicate logic". The existence of both interpretations implies that an analogy exists between data-base manipulation and programming: a same concept of logic has been both interpreted as a data-base concept and as a program-language concept. The analogy pairs "answers" with procedure declarations, negative literals with procedure calls, and queries with procedure bodies. Much earlier this analogy already made its appearance in the PLANNER family of languages, which also predated several important features of logic programming.

8. Answers which are not logical implications

Up till now we only considered correct those answers which are *logical implications* of the data base. However, not all answers which one would like to be correct are logical implications. Take for example the sentence DB = TAKES $\cup$ TEACHES (see tables 3.3 and 3.4), according to which Cook takes all courses taught by Glotz, that is,

$$\forall y \,.\, \text{Teaches}(\text{J.F:Glotz},y) \supset \text{Takes}(\text{T.L:Cook},y) \quad \ldots(8.1)$$

The query to check whether indeed Cook takes all courses taught by Glotz would have to be the negation in clausal form of the above implication:

$$P = \{+\text{Teaches}(\text{J.F:Glotz},y_o),\ -\text{Takes}(\text{T.L:Cook},y_o)\}$$

where y_o is a constant which does not occur in DB. In order for a refutation of DB $\cup \{P\}$ to exist, the rule of inference has to be applied with $-\text{Takes}(\text{T.L:Cook},y_o)$ as one parent, which is not possible. Because of the completeness of the refutation procedure, it follows that DB $\cup \{P\}$ is consistent.

A semantical reasoning to show consistency of DB $\cup \{P\}$ may be more enlightening. DB is a definite sentence and hence has a minimal model, say I_o. In I_o J.F:Glotz teaches only Math!129 and Math!225, and T.L:Cook takes only Math!129 and Math!225. Let $I_1 = I_o \cup \{\text{Teaches}(\text{J.F:Glotz},y_o)\}$. In I_1, which is also a model of DB, J.F:Glotz teaches y_o but T.L:Cook does not take y_o*). Apparently, the fact (8.1) is true in some models, but not in all: this fact is not a logical implication. Why would one consider it true?

We consider the fact (8.1) to follow from DB because we implicitly assume that Glotz teaches only what is listed in DB, that is, we assume that whatever cannot be proved is false. This assumption was recognized and discussed by R. Reiter [19], who called it the "closed-world assumption". He showed that it cannot always be made, but that for a definite sentence as data base, the closed-world assumption will not lead to contradictions.

One way of showing this is to use the fact that for a definite sentence, the intersection of all models is itself a model [6]. The intersection of all models contains exactly the provable facts. To consider the intersection as an interpretation is to assume to be false whatever cannot be proved. To find the intersection itself a model means to be free of contradiction.

*) I_1 is not an interpretation of DB because y_o is not in the universe of discourse of DB. The same reasoning can be given correctly by replacing y_o by some constant of DB.

It seems that for certain queries truth in the *minimal model* is a more appropriate criterion of correctness of an answer than truth in all models. We therefore have to ensure the existence of a minimal model. Hence the restriction of a data base to definite clauses. The refutation procedure gives answers which are true in all models, hence true in the minimal model, hence correct under both criteria. In what way can answers be obtained true in the minimal, but not all models? This can be done by means of a special predicate symbol "not".

Suppose that "not" is defined in such a way that the refutation procedure gives an affirmative response to a query -not(L) whenever a query -L would give a negative response. Let us use as example the query which asks for the names of all students that take all courses taught by Glotz. Because "not" is restricted to one-literal queries we prepare by defining some intermediate concepts. We add temporarily to the data base

+A(*x)-Teaches(J.F:Glotz,*y)-not(Takes(*x,*y))

The relation retrieved by the query -A(*x) is the set of names of students who do not take some course taught by Glotz. We also add

+B(*x)-Takes(*x,*y)-not(A(*x))

The relation retrieved by the query -B(*x) is the set of names of students who take all courses taught by Glotz.

This last example is considerably less simple than the ones discussed in section 5. This situation is not restricted to our approach to information retrieval. For instance in the use of Codd's relational algebra [3 , 5] as query language, the present example is expressed by the particularly opaque "division" operation. It might seem preferable to be able to use instead a formula of logic not restricted to clausal form. Yet the findings of Thomas [23] show that users typically make the most hilarious mistakes in the choice between existential and universal quantifiers. Perhaps we have to face the fact that some queries are more complex than others and that for the complex queries it is acceptable to have to build up step by step the desired relation, much as one builds up a simple program. It may well be that in this way users find it easier to keep track of what they are doing than when required to enter a single formula with quantifiers, which they would anyway have to build up step by step on a piece of scratch paper.

In the days before data bases, when only "files" existed, information was retrieved by a special program for each retrieval. One of the improvements offered by a data base system is that exactly the right choice of information can be retrieved by entering a single formula in a query language instead of having to write a program. It seems that formulas are not always better than programs, not when, for example, the formula has to use opaque operators and the program can be in PROLOG.

9. First-order versus second-order queries

It is instructive to see what exactly makes the query discussed in section 8 more difficult to handle than the ones in section 5. Let us look at the first-mentioned query in another way. We want the set of students who take all courses taught by Glotz. Let $R_1(\alpha)$ be the relation retrieved by the query $Q_1(\alpha)$ = -Takes(α,*y), where α is a constant. Let R_2 be the relation retrieved by the query Q_2 = -Teaches (J.F:Glotz,*y). Now Q_1 and Q_2 are queries of the type discussed in section 5 (the fact that Q_1 and Q_2 each have only one literal is not essential). Their answers are logical implications, true in all models. The name of a student taking all courses taught by Glotz is a solution for α of

$$R_2 \subseteq R_1(\alpha)$$

Both R_1 and R_2 are determined by the minimal model because the corresponding answers are logical implications. The above inclusion therefore holds only in the minimal model.

Apparently the query of section 8 involves two relations such as would be retrieved by the usual kind of query and also an inclusion relation between the relations. It is this last circumstance that makes the query belong to second-order rather than first-order logic. It should not be surprising that it is not straightforward to express such queries when one has to make use of an accidental feature which happens to transcend first-order logic, such as "not" in PROLOG, or "division" in Codd's relation algebra.

The role played here by the minimal model is the same as the one played by the minimal fixpoint in Scott's method for the theory of computation. The connection has been established in [6], but there the emphasis was on computation rather than on information retrieval. To make clear that the minimal model here plays the same role as the minimal fixpoint in [6], we shall discuss here a "computational" example.

Consider the following data base

```
DB = { +Plus(0,*y,*y)
      ,+Plus(s(*x),*y,s(*z))-Plus(*x,*y,*z)
      ,+Lessorequal(0,*y)
      ,+Lessorequal(s(*x),s(*y))-Lessorequal(*x,*y)
      }
```

Composite terms without variables have the form $\underbrace{s(s(\ldots s}_{n \text{ times}}(0)\ldots) = s^n(0)$; this one represents the nonnegative integer n. A query

$$-\text{Plus}(s^2(0),s^2(0),{*}z)$$

will give an affirmative response, with answer

$$+\text{Plus}(s^2(0),s^2(0),s^4(0))$$

In this way the first two clauses of DB constitute a program for addition. Suppose we want to prove a property of the program, for instance that, whenever an answer

$$+\text{Plus}(\alpha,\beta,\gamma)$$

is returned, that then α is less than or equal to γ, according to the program for "less or equal" in DB.

Let R_1 be the (infinite) relation retrieved by Q_1 = -Lessorequal(*x,*z). Let R_2 be the relation (likewise infinite) retrieved by Q_2 = -Plus(*x,*y,*z). The property is expressed by the second-order assertion $R_1 \supseteq R_2'$, where R_2' is the projection of R_2 on the subspace spanned by the first and third coordinates.

Now

$$P = +\text{Lessorequal}({*}x,{*}z)-\text{Plus}({*}x,{*}y,{*}z)$$

means

for all x and z, x less or equal z if
    there exists a y such that x plus y equals z

P is a first-order assertion. It may be verified that P is not true in all models of DB, but that it is true in the minimal model of DB.

10. Acknowledgement

The research reported here was supported by the National Research Council.

11. References

1. J.W. de Bakker and D. Scott
A theory of programs, IBM seminar, Vienna, August 1969

2. E.F. Codd
A relational model of data for large shared data banks
Comm. ACM 13 (1970), 377-387

3. E.F. Codd
Relational completeness of data base sublanguages
R. Rustin (ed.) Data Base Systems, Prentice Hall 1971, pp. 65-98

4. A. Colmerauer
Les Grammaires de Metamorphose
University of Marseille-Luminy, 1975

5. C.J. Date
An Introduction to Database Systems
Addison-Wesley, 1975

6. M.H. van Emden and R.A. Kowalski
The Semantics of Predicate Logic as a Programming Language,
Journal ACM, 23, pp. 733-742, (Oct. 1976)

7. M.H. van Emden
Programming in resolution logic. To appear in 'Machine Representations of Knowledge' published as Machine Intelligence 8 (eds. E.W. Elcock and D. Michie) by Ellis Horwood Ltd. and John Wylie

8. D.H. Fishman and J. Minker
Pi-Representation
Artificial Intelligence 6 (1975), 103-127

9. C.C. Green
Theorem-proving by resolution as a basis for question answering systems.
Machine Intelligence 4, B. Meltzer and D. Michie (eds.), Edinburgh University Press, 1969

10. C.A.R. Hoare
Recursive data structures
Research Report STAN-CS-73-400, Dept. of Computer Science, Stanford University

11. Kanoui, Henry
Application de la demonstration automatique aux manipulations algébriques et a l'integration formelle sur ordinateur.
Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Univ. d'Aix-Marseille, 1973.

12. R.A. Kowalski
Predicate logic as programming language
Proc. IFIP 74, North Holland, 1974, pp. 569-574

13. R.A. Kowalski
Logic for problem-solving
Memo no. 75, Department of Computational Logic
University of Edinburgh, 1974

14. R.A. Kowalski
Logic and data bases
Department of Computation and Control, Imperial College, London

15. R. Kowalski and D. Kuehner
Linear resolution with selection function.
Artificial Intelligence 2 (1971), 227-260

16. D. Kuehner
Some special purpose resolution systems
Machine Intelligence 7, B. Meltzer and D. Michie (eds.),
Edinburgh University Press, 1972

17. D. Michie
Memo functions and machine learning
Nature 218 (1968), 19-22

18. M.S. Paterson and M.N. Wegman
Linear unification
IBM Research Report RC 5904, 1976

19. R. Reiter
An approach to deductive question-answering
Technical Report, Bolt, Beranek, and Newman Inc.,
Cambridge, Mass.

20. J.A. Robinson
A machine-oriented logic based on the resolution principle.
J. ACM 12 (1965), 23-44

21. P. Roussel
PROLOG: manuel d'utilisation
Groupe d'Intelligence Artificielle
Universite de Marseille-Luminy, 1975

22. D. Scott
Outline of a mathematical theory of computation
4th Annual Princeton Conference Information
Science & System (1970), pp. 169-176

23. J.C. Thomas
Quantifiers and question-asking
IBM Research Report RC 5866 (1976)

24. D.H.D. Warren
Implementing PROLOG
Dept. of Artificial Intelligence, University of Edinburgh, 1977

25. M.M. Zloof
Query by example
Proc. Nat. Comp. Conf., AFIPS Press, 44 (1975), 431-438

26. D. Warren
WARPLAN: A system for generating plans
Memo 76, Dept. of Artificial Intelligence, University of Edinburgh, 1974