COMPUTATION AND DEDUCTIVE INFORMATION RETRIEVAL*

M.H. van Emden
Department of Computer Science
University of Waterloo

Research Report CS-77-16

May 1977

*) to appear in
E. Neuhold (ed.): Formal Description of Programming Concepts
North Holland Publishing Co., Amsterdam
Proceedings of the IFIP Working Conference
St. Andrews, New Brunswick, Canada, August 1977.

COMPUTATION AND DEDUCTIVE INFORMATION RETRIEVAL

M.H. van Emden
Department of Computer Science
University of Waterloo

1. Introduction

Some form of logic is an obvious candidate to serve as formalism for expressing the contents of a computer data base as well as the queries that select information from it. Recently, logic has found a computer application which at first sight seems to have little to do with data-base problems, namely "logic programming": the activities based on R.A. Kowalski's thesis that first-order predicate logic in clausal form can be used (according to the so-called "procedural interpretation") to specify algorithms in a human-oriented way and that such specifications can be interpreted with acceptable efficiency by a suitable uniform resolution theorem-prover.

Although algorithm specification is not central to data-base problems, the insights provided by logic programming turn out to be useful. More specifically, we describe and elaborate in the present paper a companion to the procedural interpretation of logic, which we call the "data-base interpretation of logic". This interpretation can result in a system with the following practical advantages:

a) As a special case, logic reduces to the conventional relational data base, with its attendant advantages.

b) Logic in the data-base interpretation supports a powerful system of deductive information retrieval, which solves several problems in connection with conventional data bases (see section 11). The relational view is preserved, where the user sees the data base as a set of sets of tuples. In actual fact only some of the tuples are actually stored and the others are deduced on demand; we therefore call this system a *virtual* relational data base.

c) For the casual user it is important that queries have the same form as those in "query by example", which has a simple form developed especially for the casual user.

d) From the not-so-casual user's point of view, data and procedures in the system are uniform and interchangeable, as are also queries and procedure calls; in fact the data base system is a powerful program language in its own right.

The interchangeability of the procedural and data-base interpretation is interesting from a theoretical point of view: it shows that deductive information retrieval and computation, as usually modelled in program semantics, are special cases of a common formalism.

Once again, we have attempted to make the paper self-contained: the formalism is described in full and all relevant results are stated.

2. Related Work

A more powerful system for deductive information retrieval has been designed by R. Reiter [23]. 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 [15].

His research involves the design of a data base concerning all aspects of the operation of a 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.

J. Minker [21] 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 procedure 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 [10], Kowalski's [13] on logic programming, and Clark and Tärnlund's [5] on verification of logic programs.

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. A *constant* is an identifier or an integer and should be read as denoting an unstructured object. For example

Math

129

Glotz

are constants.

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 *functor* 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 experession described later) by a term. If $e_2 = e_1\theta$ is a result of applying a substitution θ to an expression e_1 then e_2 is called an *instance* of e_1; e_1 is said to be *more general* than e_2. When θ, e_1, e_2 are such that $e_1\theta = e_2\theta$, e_1 and e_2 are said to be *unifiable* and θ is said to be the *unifier*. If a unifier exists then there is also a most general unifier [25].

A composite term is to be read as denoting a structured object; the functor 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 functor 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 functors, 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 nonempty set of clauses.

A *clause* is a pair of sets of atoms written as

$$A_1,\ldots,A_n \leftarrow B_1\& \ldots \&B_m \qquad m,n\geq 0$$

or, when both sets are empty, as

□

The set $\{A_1,\ldots,A_n\}$ is the *conclusion* of the clause and $\{B_1,\ldots,B_m\}$ is the *premiss* of the clause.

An *atom* (short for atomic formula) is $P(t_1,\ldots,t_k)$ where P is a predicate symbol and where $t_1,\ldots,t_k$ 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

$$A_1,\ldots,A_n \leftarrow B_1\& \ldots \&B_m \qquad \ldots(3.1)$$

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 (or procedures) 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 or...or A_n can only be returned as an answer if the query

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

has success as response. 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 atom in the conclusion. A

definite sentence is one where all clauses are definite. For reasons to be explained later, we only consider data bases which are definite sentences.

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 a relation to hold between fully specified objects. The subset in a sentence of all variable-free clauses with one atom in the conclusion and an empty premiss 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 variable-free arguments specifies a relation in the same way as it is done by an "array" in the sense of Codd [6].

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 [23], where the collection of arrays is called the extensional data base and the rules are called the intensional data base.

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

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

The definite clauses with an empty premiss 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, namely, 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.

COMPUTATION AND DEDUCTIVE INFORMATION RETRIEVAL

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

```
{ 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
	C :Twill	Math!170

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

Table 3.4

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

4. Semantics for Logic in Clausal Form

The set of variable-free terms that contain only constants or other functors 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 the universe of atoms. An interpretation I is said to be a *model of* S iff S is true in I.

A sentence is *true in I* iff all of its clauses are true in I.

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

A variable-free clause is *true in I* iff at least one of the atoms in its conclusion is true in I or at least one of the atoms in its premiss is not true in I.

A variable-free atom A in a conclusion is *true in I* iff $A \in I$.

A variable-free atom B in a premiss is *true in I* iff $B \notin I$.

The informal semantics given before conform to this definition of truth. It may be useful to note that a clause

$$\leftarrow B_1,\ldots,B_m$$

is true in I iff at least one B_i is not true I for each variable-free instance of the clause. The clause may therefore be read as the negation of

$$\text{there exist } y_1,\ldots,y_k \text{ such that } B_1 \text{ and}\ldots\text{and } B_m$$

A sentence is *consistent* iff it is true in at least one interpretation. We will only use *logical implication* as an informal notion explained in terms of consistency:

$$S \cup \{\leftarrow A_1,\ldots,A_n\} \text{ is inconsistent}$$

is to be read as

$$S \models \text{ there exist } x_1,\ldots,x_k \text{ such that } A_1 \text{ and}\ldots\text{and } A_n$$

where $\models$ stands for "logically implies" and $x_1,\ldots,x_k$ are the variables in $A_1,\ldots,A_n$.

With a definite sentence S there is associated a transformation T from interpretations to interpretations defined as follows:

$T(I)$ contains a variable-free atom A iff there exists a variable-free instance $A \leftarrow B_1,\ldots,B_m$ of a clause of S such that $B_1,\ldots,B_m \in I$, $m \geq 0$.

A useful characterization of truth in I for a definite sentence S is that $I \supseteq T(I)$ iff S is true I, where T is the tranformation associated with S. It may be shown that a definite sentence has a least model, that is, one contained in all of its models [9].

5. Inference

As soon as it is possible for an information retrieval system to produce answers not explicitly present in the data base, it becomes important to know in what sense such answers are justified by the contents of the data base. In the data base interpretation of logic we require, as a first approximation, answers to be logical implications of the data base when the latter is regarded as a sentence of first-order predicate logic. We will see that an answer to a query is elicited by a refutation procedure showing the query to be inconsistent with the data base and that the answer so produced is a logical implication.

Numerous refutation procedures for sentences in clausal logic have been based on J.A. Robinson's resolution principle [25]. For our purpose SL-resolution [17] is most important and especially a variant [18,14] intended for use with sentences containing, apart from one negative clause, only definite clauses. Because of this restriction we will refer to this resolution refutation procedure as SLD resolution: SL-resolution for Definite clauses.

SLD-resolution conceptually constructs a sequence (a *refutation*) $S_0,S_1,\ldots$ of sentences such that each successor has the same set of models as its predecessor in the sequence. Each successor is said to be obtained by *resolution* from its predecessor. If some S_n in the sequence contains the empty clause, it follows that it is inconsistent and that thereby S_0 has been refuted. SLD-resolution incorporates a *selection rule*, which designates a unique atom (the *selected atom*) in any negative clause. S_j is obtained by replacing in S_{j-1} a negative clause (one *parent* in the resolution)

$$\leftarrow A_1,\ldots,A_i,\ldots,A_n$$

with selected atom A_i, by

$$\leftarrow (A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n)\,\theta \quad \text{with} \quad 1 \leq i \leq n$$

if S_{j-1} contains a clause (the other parent in the resolution)

$$A \leftarrow B_1,\ldots,B_m$$

such that A and A_i are unifiable with most general unifier θ.

If S_0 contains, apart from one negative clause, only definite clauses, then the same holds for all S_j, $j \geq 0$. The correctness of SLD-resolution says that the presence of the empty clause in some S_n implies inconsistency of S_0. The completeness of SLD-resolution, which is independent of the selection rule, says that inconsistency of S_0 (having one negative clause and definite clauses) implies the existence of a refutation.

6. The Procedural Interpretation of Logic

According to Kowalski's procedural interpretation [13], several key concepts of programming in a high-level language can be interpreted in terms of logic. A summary follows.

Procedure definition:

A definite clause of which the conclusion is interpreted as the procedure heading and the premiss as the procedure body. The arguments of the conclusion are the formal parameters. The variables of the premiss which do not occur in the conclusion are the local variables.

Statement:

The only kind of statement which is a procedural interpretation, is the sequence of procedure calls. A premiss is interpreted as a sequence of procedure calls. A negative clause is an initial sequence of procedure calls. The replacement of the call by the body of a procedure definition is resolution. The replacement of formal by actual parameters is unification.

Output:

The product of the successive unifications in a refutation determines a substitution of terms for the variables in the initial procedure call. These terms are the output of the computation.

Interpreter:

The SLD-resolution procedure with a selection rule that regards negative clauses as ordered sets of atoms and selects the first atom.

The procedural interpretation reduces logic to a very special case. Often a less drastic reduction yields a useful generalization of the usual high-level procedure-oriented program languages. Some of these generalizations have already been advocated independently of the procedural interpretation as piecemeal "advanced features". Examples of such features, or of aspects of the same feature, are: recursive data structures [11], pattern matching, indeterminate procedure call by pattern matching, success versus failure as computation result, automatic backtracking upon failure [4]. A particularly important feature, also found in MICROPLANNER [4], is the possibility of empty procedure bodies so that the arguments of the heading can act as a tuple in the relation named by the procedure identifier. It is this aspect of the procedural interpretation of logic which gives rise to the data-base interpretation of logic. It is the basis of our unified model of computation and deductive information retrieval.

7. The Data-base Interpretation of Logic

According to the data-base interpretation of logic, the key concepts connected with data bases may be described in terms of logic as follows.

Virtual relational data base:

A set DB of definite clauses. In case DB contains arrays only, it is a conventional relational data base.

Query:

A clause $\leftarrow A_1 \,\&\ldots\&\, A_n$ $(n \geq 0)$ with an empty conclusion. Note that such

a clause may also be read as a negation (Section 4).

Response:

The reaction of the retrieval procedure to a query Q with a given data base DB is called the *response*, which can be success or failure. *Success* occurs when the sentence DB ∪ {Q} is inconsistent. *Failure* occurs when the retrieval procedure determines that DB ∪ {Q} is not inconsistent. If DB contains arrays only, a response is guaranteed to occur. Consistency is not decidable for arbitrary definite sentences as DB, so that a response does not necessarily occur.

Answer to a query:

If the response is success, then the query has an answer. The refutation procedure has determined not only that DB ∪ {Q} is inconsistent, but also a substitution θ such that DB ∪ {Qθ} is inconsistent. Suppose now that $Q = \leftarrow A_1 \,\&\ldots\&\, A_n$; the *answer to* Q is the sequence $A_1\theta \leftarrow,\ldots,A_n\theta \leftarrow$ of clauses. Their relationship to DB is

$$DB \models \text{for all } x_1,\ldots,x_k,\ A_1\theta \text{ and}\ldots\text{and } A_n\theta$$

where $x_1,\ldots,x_k$ $(k \geq 0)$ are the variables of the answer $A_1\theta\leftarrow,\ldots,A_n\theta\leftarrow$.

A clause of the answer may or may not occur in DB. In the latter case such a clause has been deductively retrieved; its presence in DB would be redundant.

Relation retrieved:

In case the response to a query is success, there may be more than one substitution θ as described above. Each such θ determines a k-tuple of (not necessarily variable-free) terms substituted for $x_1,\ldots,x_k$, $k \geq 0$, the variables of the query. The set of the variable-free instances of these k-tuples is the *relation retrieved* by the query.

Retrieval procedure:

Resolution refutation procedure for first-order predicate logic in clausal form. In the following examples we assume that the retrieval procedure is SLD-resolution.

Note that an $S_0 = DB \cup \{Q\}$ contains one clause, namely Q, with an empty conclusion and the conclusions of all its other clauses (those of DB) contain exactly one atom. With such an S_0 all sentences S_i of a refutation have the same property, in fact, $S_i = DB \cup \{Q_i\}$ with Q_i a query. 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. If DB only contains arrays then the retrieval procedure only returns unmodified clauses as answer.

Example 7.1

Let DB = TAKES ∪ TEACHES ∪ SCHEDULED ∪ STUDENT ∪ RULES as given in Tables 3.1,...,3.4.
Let $Q_0 = \leftarrow$ Takes(*x,Math!129).

The rule of inference will use as parents Q_0 and

Takes(M:Adiri,Math!129) ← in TAKES

and produce $Q_1 = \square$, the empty query. The unifier substitutes M:Adiri for *x. Thus, the *response* is *success*, 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_0 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_0 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 in the sense that none of the tuples, are stored; all are simulated by 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 success as response when

DB = TAKES ∪ TEACHES ∪ SCHEDULED ∪ STUDENT ∪ 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 ∪ {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 (here a singleton) of variables in the query, namely

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

Example 7.2

It may happen that all available information is already in the query. In such a case the distinction between response and answer is particularly useful. With DB as above and query

←Takes(M:Adiri,Math!129)

the response is success and the relation retrieved contains the 0-tuple as only element.

8. Computation and Inductive Information Retrieval

The relational model is not only useful for representing data bases and queries to them, but also for computation. Consider the following example of a data base intended to answer queries about the results of appending lists:

{append(nil,nil,nil) ←

,append(a.nil,nil,a.nil) ← , ...

,append(b.nil,nil,b.nil) ← , ...

.

.

.

}

Table 8.1

The lists of this example are "nil" if empty, and $\alpha \cdot \beta$ otherwise, where the alpha is a list element, the beta a list, and the period is a functor which is also an infix operator.

Except for the constructed data as entries in the tuples, this sentence corresponds to an array of a conventional relational data base. With the data-base interpretation of logic the following less redundant data-base can be used to obtain the same effect:

```
{ append(nil,*y,*y) ←
,append(*u·*x,*y,*u·*z) ← element(*u)
                          & append(*x,*y,*z)
,list(nil)
,list(*x·*y) ← element(*x) & list(*y)
,element(a) ← , element(b) ← , element(c) ←
,element(d) ←
}
```

Table 8.2

A query

```
← append(a.b.nil,c.d.nil,*z)                    ...(8.1)
```

has the same result with Table 8.1 as with Table 8.2. The same holds for the queries

```
←append(*x,c.d.nil,a.b.c.d.nil)
          (response:success; answer:*x = a.b)
←append(*x,b.d.nil,a.b.c.d.nil)
          (response:failure)
←append(*x,c.*y,a.b.c.d.nil)
          (response:success; answer:*x = a.b.nil
                                *y = d.nil
          )
←append(a.b.nil,c.d.nil,a.b.c.d.nil)
          (response:success; answer:        )
```

In answering the query (8.1), SLD-resolution as retrieval procedure behaves in a way similar to a typical LISP processor interpreting the usual recursive definition of append for LISP-lists. The other queries illustrate the power of the relational formalism, where an argument is not restricted to input or output. The example shows that in logic programming the difference between computation and information retrieval is one of degree rather than in kind. In example 7.1 table look-up is the dominant operation, hence we have information retrieval. Here deduction is dominant, and computation seems to be a more appropriate description. Logic programming may be viewed as information retrieval on a virtual relational

data base without any built-in bias towards either look-up or deduction. Michie's idea of a memo-funtion [20] 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 non-termination in computation, and more generally in the application 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, as in the de Bakker/Scott "relational theory" of programs [2]. Because arguments in the procedural or data-base interpretation of logic are not restricted to input or output, binary relations do not have the same special status. In fact, interpretations I, satisfying $I \supseteq T(I)$ (see Section 4), play the role in logic programming of the binary relations in the relational theory of programs.

9. Answers which are not true in all models

Up till now we considered correct only those answers which are true in all models of the data base. However, this is not the case for all answers which one would intuitively consider correct. Take for example the sentence DB = TAKES ∪ TEACHES (see Tables 3.3 and 3.4), according to which Cook takes all courses taught by Glotz, that is

$$\text{for all } y,\ \text{Teaches(J.F:Glotz,}y\text{)} \quad \text{implies Takes(T.L:Cook,}y\text{)} \qquad \ldots(9.1)$$

DB is a definite sentence and hence has a least model, say I_0. In I_0, J.F:Glotz teaches only Math!129 and Math!225 and T.L:Cook takes only Math!129 and Math!225. Let $I_1 = I_0 \cup \{\text{Teaches(J.F:Glotz,Math!170)}\}$. In I_1, which is also a model of DB, J.F:Glotz teaches Math!170 but T.L:Cook does not take it. Apparently, (9.1) is true in some models, but not in all: it is not a logical implication.

Why should one want answers to be able to take into account facts like (9.1)? Before discussing two possible points of view on this question, it should be noted that the query

$$\leftarrow \text{Teaches(J.F:Glotz,Math!170)}$$

gives failure as response. If only we could interpret this failure as falsity, our database interpretation of logic would also be able to tell us that Cook takes all courses taught by Glotz.

According to one point of view the required additional information should be expressed in the data base itself. According to the other point of view the criterion of correctness, which requires an answer to be true in all models, is too stringent. Let us first discuss the latter alternative, according to which one model of the typically many models of a definite sentence as data base is assumed: queries are correct if they reflect the situation in this particular

model. Definite sentences have the property that the intersection of all models is itself a model, the least model, as was mentioned in Section 4. The least model reflects the assumption according to which, for example, Glotz teaches *only* what is listed as such in the array TEACHES(Table 3.4). The intersection of all models is the set of provable facts [9]. To consider the intersection as interpretation is to assume (usually called "the closed-world assumption") false whatever can be established as unprovable. That the intersection is itself a model implies that the closed-world assumption will not lead to contradiction. We owe to Reiter [23] the observation that the closed-world assumption is safe to make for definite sentences.

SLD-resolution as retrieval procedure gives us all answers which are logical implications, i.e. true in all models, hence true in the least model. If the least model is taken as criterion of correctness, then it is necessary to supplement SLD-resolution with an inference mechanism that yields answers true in the least model, but not necessarily in all.

According to the other point of view the missing information should be supplied by the data base. In our data-base interpretation the array TEACHES of Table 3.4 only states that

```
for all  x,  Teaches(J.F:Glotz,x)

                 if(x = Math!129
                   or x = Math!225
                   )

and for all  x,  Teaches(C:Twill,x)
                 if x = Math!170
```

which does not exclude that Glotz possibly teaches another course as well. An obvious remedy is to change the "if" to "iff" and then from

```
not(Math!170=Math!129  or  Math!170=Math!225)
```

we conclude

```
not Teaches(J.F:Glotz,Math!170)
```

However, such "iff" definitions expressed in clausal form do not yield a definite sentence. The advantages of the procedural interpretation of clausal logic (as developed up till now) depend on the use of SLD-resolution which requires definite clauses for proper operation. A promising approach, being investigated at present by R.A. Kowalski and K.L. Clark at Imperial College, is to find such a formulation of "iff" definitions that, when each "iff" is replaced by an "if", a definite sentence in clausal form results. SLD-resolution, when used as retrieval procedure/program interpreter, only sees this definite sentence. The formulation of the "iff"-definitions has the important property that whenever the retrieval procedure gives failure as response to a query, the query itself, regarded as a negation, is a logical implication of the "iff"-definitions. The required formulation of the iff-definitions would thus justify interpreting failure as falsity.

In the practice of logic programming the query asking for students who take all courses taught by Glotz is expressed as follows. Suppose the refutation procedure is programmed in such a way that it gives success as response to a query ← not(L) whenever the query ← L would give failure as response. Let us use as example a query which asks for the names of all students each of which takes all courses taught by Glotz. Because it is convenient to restrict "not" to queries with one atom, we add temporarily to the data base

```
A(*x) ← Teaches(J.F:Glotz,*y) & not(Takes(*x,*y))
```

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

$$B(*x) \leftarrow Takes(*x,*y) \;\&\; not(A(*x))$$

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

This use of "not" interprets failure as falsity, but happens to conform also to the point of view which selects the least model as criterion of correctness, because success as response to a query $\leftarrow not(L)$ implies that all variable-free instances of an atom L are false in the least model.

10. Properties of Logic Programs

Program properties are often not logical implications of the logic programs to which they pertain. They have this in common with the answers to the queries discussed in the previous section. With respect to program properties two approaches are again possible: the least model of the program is the criterion of correctness or a strengthened version of the program is used to obtain as a logical implication the property to be proved. The first approach can use in principle [9] the methods of program verification based on least fixpoints, but as far as we know the principle has not been worked out. The second approach has been successfully followed by Clark and Tärnlund [5] to examples considerably more realistic than the one below.

A query

$$\leftarrow append(\alpha,nil,\alpha)$$

to Table 8.2 will give success as response for any list α. Hence it is desirable to be able to regard

$$\text{for all } x,\ list(x) \text{ implies } append(x,nil,x) \qquad ...(10.1)$$

as a property of the logic program in Table 8.2 for computing the append relation. Yet it is easy to see that (10.1) is not true in all models of the sentence in Table 8.2, and hence not a logical implication, although (10.1) is true in the least model. We saw a similar situation with the query for all students each of which takes all courses taught by Glotz: (10.1) can be proved by strengthening the definition (8.2) or by inference for least models. Note that in the "relational theory" of de Bakker and Scott [2,1,3], among many other publications, the input-output relation computed by a program is characterised as the least solution x of $x = Tx$ or of $x \supseteq Tx$. The choice is immaterial because both have the same least solution. Scott's induction rule is justified for the least solution of $x = Tx$. It may be useful to investigate in the de Bakker/Scott relational theory whether program properties can also be derived as applicable to all solutions of $x = Tx$, and whether such derivations have advantages over those using Scott's induction rule.

Let us consider an example, similar to one of Clark and Tärnlund [5], of a strengthened version of the sentence in Table 8.2 such that property (10.1) holds in all models. The sentence in Table 10.1 is not in the clausal form of logic. The main difference with Table 8.2 is that "if" has been replaced by "iff".

$$\forall x,y,z.\mathrm{append}(x,y,z) \text{ iff } [x=\mathrm{nil} \ \&\ y=z \text{ or } \exists u,x',z'.\ x=u.x' \ \&\ z=u.z' \ \&\ \mathrm{element}(u) \ \&\ \mathrm{append}(x',y,z')]$$

$$\&\ \forall x.\mathrm{list}(x) \text{ iff } [x=\mathrm{nil} \text{ or } \exists u,x'.\ x=u.x' \ \&\ \mathrm{element}(u) \ \&\ \mathrm{list}(x')]$$

$$\&\ \forall x.\mathrm{element}(x) \text{ iff } [x=a \text{ or } x=b \text{ or } x=c \text{ or } x=d]$$

Table 10.1

It may be shown that (10.1) is true in all Herbrand models of Table 10.1, which is apparently a sufficiently strengthened version of Table 8.2.

According to the other approach we only try to show (10.1) in the least model of Table 8.2. One method among several is to show that the following induction schema for lists is valid in the least model:

$$[P(\mathrm{nil}) \ \&\ [\forall u,x.\mathrm{element}(u) \ \&\ P(x) \rightarrow P(u.x)]] \rightarrow [\forall x.\mathrm{list}(x) \rightarrow P(x)]$$

where $P(\alpha)$ stands for a formula with α as free variable

With the induction schema and the nonclausal version of the sentence in Table 8.2, (10.1) can be shown to hold by verifying the induction schema with $P(\alpha)$ replaced by $\mathrm{append}(\alpha,\mathrm{nil},\alpha)$.

11. Towards an integrated data-base and programming system

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 [7,26,19], 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 [7], robot plan formation [27], symbolic mathematical computation [12], and in compiler writing [7]. Recent implementations of PROLOG [28,24] 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. Such a system is useful because it has the

following advantages over conventional data bases:

a) The user does not have to know which relations are primitive. For example, in a conventional data base on family relationships the choice must be made between storing the "Parent" relation or its inverse, the "Child" relation. A user will have to remember the arbitrary choice, but not when deductive information retrieval is available: when the arbitrary choice has been made to store the "parent" relation as an array, the rule

$$\text{Child}(*x,*y) \leftarrow \text{Parent}(*y,*x)$$

can be added, and the user may express his query using either "Child" or "Parent".

b) As has been pointed out by Kowalski [15], relations stored as large arrays are often highly redundant: they are typically a consequence of an organization's regulations or policy expressed as rules, exceptions to the rules, exceptions to the exceptions, but usually not much further. In our approach to data base design, such rules and exceptions are directly expressed in logic and consequences to be retrieved are deduced on demand. As a result our virtual relational data base is simpler to understand and to check for compliance with regulation or policy. A virtual relational data base is often more and never less compact. An array of a conventional relational data base is often like a table of logarithms, which can be advantageously replaced by a compact subroutine computing on demand the required values.

c) The data retrieved by queries to a data base are often not required for inspection by humans, but serve as input for programs, for example for statistical analysis. Such usage is facilitated when there is one language for retrieval and processing.

d) Codd's query languages [8] are too difficult for most casual users. Zloof's "query-by-example" [29] has been shown to be an improvement in this respect. For some important classes of queries ("projections" and "joins" in Codd's terminology), our way of expressing queries as clauses is virtually identical to "query-by-example".

12. Acknowledgements

Discussions with R.A. Kowalski were essential in the development of the material presented here. The National Research Council provided financial support for this research.

13. References

1. J.W. de Bakker and W.P. de Roever: A calculus of recursive program schemes; in M. Nivat (ed.): Automata, Languages, and Programming. North Holland, Amsterdam, 1973.

2. J.W. de Bakker and D. Scott: A theory of programs, IBM Seminar, Vienna, August 1969.

3. A. Blikle: An algebraic approach to the mathematical theory of programs; CCPAS Report 119, Computation Centre, Polish Academy of Sciences, Warsaw, 1973.

4. D.G. Bobrow and B. Raphael: New programming languages for AI Research; Computing Surveys, 6 (1974), 153-174.

5. K.L. Clark and S.A. Tärnlund: A first-order theory of data and programs; Proc. IFIP 77.

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

7. A. Colmerauer: Les grammaires de metamorphose; in L. Bolc(ed.): Natural Language Communication with Computers, Springer Lecture Notes in Computer Science, 1977.

8. C.J. Date: An Introduction to Data-Base Systems; Addison-Wesley, 1975.

9. M.H. van Emden and R.A. Kowalski: The semantics of predicate logic as a programming language; J.ACM 23 (1976), 733-742.

10. C.C. Green: Theorem-proving by resolution as a basis for question answering systems; in B. Meltzer and D. Michie (eds.), Edinburgh University Press, 1969.

11. C.A.R. Hoare: Recursive data structures; STAN-CS-73-400, Dept. of Computer Science, Stanford University.

12. H. Kanoui: Application de la demonstration automatique aux manipulations algebriques et à l'integration formelle sur ordinateur; Group d'Intelligence Artificielle, U. d'Aix-Marseille II.

13. R.A. Kowalski: Predicate logic as a programming language; Proc. IFIP 74, North Holland, 1974, 556-574.

14. R.A. Kowalski: Logic for problem-solving; Memo 75, Dept. of Computational Logic, University of Edinburgh, 1974.

15. R.A. Kowalski: Logic and data bases; Dept. of Computation and Control, Imperial College, 1976.

16. R.A. Kowalski: Algorithm = Logic + Control; Technical Report, Dept. of Computation and Control, Imperial College, 1977.

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

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

19. H. Meloni: PROLOG, mise en route de l'interpreteur et exercises; Groupe d'Intelligence Artificielle, U. d'Aix-Marseille II, 1976.

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

21. J. Minker: Performing Inferences over relational data bases; TR-363: Dept. of Computer Science, University of Maryland, 1975.

22. J. Minker: Set Operations and Inferences over Relational Data Bases; TR-427, Dept. of Computer Science, University of Maryland, 1975.

23. R. Reiter: An approach to deductive questions-answering; Technical Report, Bolt, Beranek, and Newman Inc., Cambridge, Mass.

24. G.M. Roberts: An implementation of PROLOG; M.Sc. Thesis, Dept. of Computer Science, University of Waterloo, 1977.

25. J.A. Robinson: A review of automatic theorem-proving; pp. 1-18 in: Proc. Symp. App. Math. Vol XIX, J.T. Schwartz (ed.), Am. Math. Soc., Providence, R.I., 1967.

26. P. Roussel: PROLOG, manuel d'utilisation; Groupe d'Intelligence Artificielle, U. d'Aix-Marseille II, 1975.

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

28. D.H.D. Warren: Implementing PROLOG; Dept. of Artificial Intelligence, University of Edinburgh, 1977.

29. M.M. Zloof: Query by Example; Proc. Nat. Comp. Conf., AFIPS Press, 44 (1975), 431-438.