DATABASE INFERENCING FOR DECISION SUPPORT

Ronald M. Lee

April 1983
WP-83-47

INTERNATIONAL INSTITUTE FOR APPLIED SYSTEMS ANALYSIS
2361 Laxenburg, Austria

ABSTRACT

The use of databases for management decision support requires flexible inferencing mechanisms. The use of logic programming for these purposes is explored. To be flexible, however, this requires the logical decomposition of the database into elementary predicates.

CONTENTS

DATABASE INFERENCING FOR DECISION SUPPORT

Ronald M. Lee

A. INTRODUCTION

Database management applications have become common in nearly all types of private and public organizations. Yet, their use for higher level managgement dicision making has been limited. One reason for this is the lack of inferencing mechanisms to provide the higher levels of abstraction needed in various decision contexts.

This paper explores the use of logic programming for these purposes. It concludes with the observation that database inferencing, to be flexible, requires a (conceptual, logical) decomposition of the database into elementary predicates.

B. DATABASE MANAGEMENT

Database management (DM) arose originally from a need for a specialization of labor in data processing. Applications programmers had the dual function of satisfying user requirements as well as efficiently maintaining the data on various storage devices.

As long as applications tended to be relatively independent, this was not a great problem. However, as more and more data files came to be shared among various applications, coordination problems arose. Different applications favored different types of data organization.

Database Management Systems (DBMSs) offered a separation of these concerns. Essentially, a DBMS translates between an abstracted view of data, accessed by application programs, and its actual physical representation.* What the appropriate abstracted view should be, so-called 'data models', became an interesting research question and has been the subject of prolonged debate for nearly a decade. The basic camps, eventually, centered around a graphical view called the Network Model as opposed to a tabular view, the Relational Model. (Date, 1977, gives a good comparison.) While the two views are closely compatible, the Network Model seems to have certain advantages from the user engineering standpoint, and has been more widely implemented. The Relational Model, on the other hand, is mathematically simpler, and for that reason has been the more favored view in research discussions. The Relational Model is also adopted here as representing the database management paradigm.

* The abstraction process may actually go a step further as recommended by the ANSI/X3/SPARC report (Tsichritsis and Klug, 1977). Following that report, programs would access an 'external view' of the data, which is a subset of a master view called the 'conceptual schema'. This in turn is mapped to the 'internal schema' indicating actual physical storage.

C. THE RELATIONAL DATA MODEL

The Relational Model was originally proposed by Codd (1970). In this view, data items are regarded as arranged in rectangular tables consisting of columns and rows. Columns are called *attributes*, rows are called *tuples*, while the entire table is called a *relation*. An example relation, containing data on employees, is the following:

EMPLOYEE	(ID#,	NAME,	RANK,	SALARY)
	12	JONES	CLERK	10000
	51	SMITH	CLERK	10000
	27	DOE	MANAGER	25000
	05	ELIOT	PRESIDENT	50000

Note that rows correspond to individual employees whereas the columns indicate the various recorded features of the employee. This is the general convention, i.e. that rows correspond to individuals in the environment ('instances') while columns indicate their attributes. In the EMPLOYEE relation, the attribute ID# (identification number) is a 'key attribute', that is, a unique identifier (of the individual in the environment corresponding to the tuple). Such keys serve as cross references to other relations, such as in the following relation, showing superior/subordinate relationships.

WORKS-FOR	(SUPERIOR#,	SUBORDINATE#)
	27	12
	27	51
	05	27

In this case, both SUPERIOR# and SUBORDINATE# refer to ID# data items in the EMPLOYEE relationship. The identifying key for the WORKS-FOR relation is however the conjunct of the SUPERIOR# and SUBORDINATE# attributes.

In the theory behind the Relational Model, database relations are regarded as mathematical relations over various domains of data items. An important concept in this theory is the so-called 'functional dependency' that may arise between attribute domains. That is, if one attribute, A, is functionally dependent on another, B, then an update to B requires a corresponding update to A.

In the above example, for instance, it may be the case that salary depends on rank. That is, each rank has a fixed salary. Hence, knowing an employee's rank, we can determine his or her salary. In this case, the database would be redundant, since the salaries of clerks are recorded twice. To avoid potential inconsistencies (e.g. having one clerk's salary different than another's) the database should be normalized so that each such fact is recorded only once. In this example, the EMPLOYEE relation would be divided into two relations, EMPLOYEE and PAY-SCALE, as shown below. (For further discussion on normalization, see Codd, 1972, Fagin, 1977.) Note that in the PAY-SCALE relation, the attribute RANK serves as

the identifying key.

EMPLOYEE	(ID#,	NAME,	RANK)
	12	JONES	CLERK
	51	SMITH	CLERK
	27	DOE	MANAGER
	05	ELIOT	PRESIDENT

PAY-SCALE	(RANK,	SALARY)
	CLERK	10000
	MANAGER	25000
	PRESIDENT	50000

However, this decomposition is appropriate only *if* the organization's personnel policy makes salary a unique function of rank. The equal salaries of the two clerks may only have been an accidental coincidence, not due to a functional dependency. This is a fundamental point: functional dependencies cannot be detected from patterns in the actual data alone. They reflect relationships between *possible* values of attributes.

This is due to the fact that organizational databases are dynamic, that is, they are continually being updated reflecting the effect of organizational transactions such as sales, inter-departmental transfers, production runs, etc. If the database were completely static, functional dependencies could be detected from the actual data, but then they would not be of interest; since there are no updates, no accidental inconsistencies could arise.

D. INFERENCING ON DATABASES

The major use of DM databases to date has been in data processing applications; hence mainly for structured, operational level activities such as sales order processing, billing and inventory control. These applications are characterized by high volumes of routine transactions. Performance criteria are mainly speed and efficiency. Databases might also be useful in less structured, longer range activities, though the requirements in this case are somewhat different:

a. information is usually required in more summarized form

b. access is less routine — information must be retrievable in a variety of forms and combinations

c. the information is often used in combination with other informational and computational resources.

These are criteria for using DM databases in *decision support* applications. The principle point is that the data needs in these cases, though contained in the database, will often not be at the detail level nor in the structural arrangement in which the database was designed. It is for these uses that a mechanism providing inferencing on the database is needed.

One obvious way of summarizing data is simple arithmetic calculations — e.g. counts of inventory. Lacking however is a corresponding framework of qualitative inferencing. For instance, if you have an inventory of three apples and two oranges and count them up, you have five 'things', but what descriptive label should be attached to this broader class? In this case a system of qualitative inference is needed. More

realistic examples abound, e.g. in accounting data if you have $500 in cash and $700 in accounts receivable, then you have $1,200, but of what? Conversely, one might wish to make a query about the quick assets of the company when the database only contained data on cash and accounts receivable.

E. PREDICATE CALCULUS AND LOGIC PROGRAMMING

Further discussion of database inferencing for decision support applications requires a brief background on predicate logic and its computational counterpart, logic programming.

1. Predicate Calculus

It is assumed that the reader is at least generally familiar with the first order predicate calculus (FOPC) and its syntax. The following is thus only a review.

The description of a logical system begins by declaring its *universe of discourse*. In a propositional (zero order) logic, this amounts to a set of statements (propositions) asserted to be true. In a first order logic, a separation is made between individual entities (or just *individuals*), and the properties and relationships to other individuals. The latter are indicated, respectively, by one and n-place *predicates*. For a first order logic the domain of discourse is called the *domain of individuals*. (For the moment, the individuals described by the logic can be imagined as discrete physical objects at a point in time.) In summary form, the basic constructs of a first order predicate calculus are as follows:

1. *Propositions.*

   These are complete logical statements having a truth value. These are indicated symbolically by capital letters —e.g. P,Q,R.

2. *Logical connectives.*

   These combine one or more propositions to form new logical statements, also having a truth value. The logical connectives

used here are as follows:

↔	equivalence
→	implication
&	conjunction
V	disjunction (inclusive)
W	disjunction (exclusive)
~	negation

3. *Individual constants and variables.*

These stand for objects in the domain of discourse — e.g. individual trucks or employees.

Individual constants are denoted as one or more upper case letters, possibly containing non-leading digits or hyphens; e.g. A, GEORGE, TRUCK-7.

Individual variables are denoted as either lower case letters, e.g. x, y, z, or as a "?" followed by one or more capital letters or digits, e.g. ?ID, ?SALARY. (The dual notation here is a compromise between the logical convention of variables as lower case letters, and the database management convention of capitalizing names of attributes that are recognized as variables in a logical interpretation.)

4. *Functions.*

These map one or more individuals to another — e.g. SUPERVISOR (JONES) refers to another individual who is Jones'

supervisor. Functions may take zero or more arguments and always result in a reference to a single individual. Functions may thus appear wherever an individual constant is allowed. Indeed, a zero-place function is the same as an individual constant. Functions are therefore denoted in the same way as individual constants, but followed by an argument list, e.g, F(A), BOSS(SMITH)

5. *Predicates.*

These indicate features, properties, attributes, etc., applied to zero or more individuals. Predicates will be denoted by upper case letters or words, e.g. P(x), RED(?X), OWN(x,y). When a predicate is applied to individual constants or to quantified individual variables (see below), or to functions of these, it has a truth value and may be combined to form other logical statements using the logical connectives above. A zero-place predicate is equivalent to a proposition.

6. *Logical quantifiers.*

These indicate the range of individual variables. The principal ones are:

$\forall x$ universal quantifier
(for all x, for each x, —
ranging over all individuals
in the universe)

$\exists x$ existential quantifier

(for some x — ranging over

at least one individual)

Parentheses are used in the usual fashion.

2. Logic Programming

Mechanical theorem proving in the predicate calculus has been a central area of AI research since its outset. As with logic generally, the original goal was to reproduce mathematical reasoning. Thus, an early success was the Logical Theorist program by Newell, Shaw and Simon (1963), which reproduced the proofs of Russell and Whitehead's *Principia Mathematica*. Indeed, the program found several original proofs of certain theorems. A more recent success is the AM* program of Lenat (Davis and Lenat, 1982). The goal in AM is not only to prove specified theorems from a given set of axioms, but also to decide for itself which axioms are interesting to prove. It thus is a model of mathematical discovery.

Just as modern logic is now used to formalize reasoning in non-mathematical subjects, AI theorem-proving systems have also been applied to model reasoning in other areas. Basic axioms about the world are asserted and the system deduces further statements (theorems) based on these axioms.

Whereas mechanical theorem-proving for the propositional calculus is relatively easy, theorem-proving in the (first order) predicate calculus

* Lenat: "the original meaning of this mnemonic has been abandoned. As Exodus states, 'I *AM* what I *AM*'." (Davis and Lenat, 1982, p. 3).

is computationally much more difficult. One problem is that there are typically a number of inference rules available, corresponding for example to different arrangements of leading quantifiers or different combinations of logical connectives. While these are a convenience to human logicians, they lead to excessive branching and an extremely large search space for mechanical proofs.

The so-called 'resolution method' of Robinson (1965) offers considerable computational simplification by reducing logical assertions to an elementary 'clausal' ('Horn clause') form. In this form, only one inference rule, resolution, is needed. (Resolution essentially combines the inference rules of modus ponens and substitution.) Assertions in clausal form have the following general pattern:

$$P_0 \leftarrow P_1 \,\&\, P_2 \,\&\, \ldots \,\&\, P_n.$$

where the P_i are predicates of the form $P(x_1, x_2, \ldots, x_k)$. This can be read: "to prove P_0 it is sufficient to prove P_1, P_2, ..., and P_n. All variables are assumed to be universally quantified. It can be shown* that any first order assertion can be reduced to this form. The resolution method provides the basis for a family of theorem-proving languages that together have come to be known as 'logic programming'. The best known among these is the language PROLOG (abbreviating PROgramming in LOGic), originally invented by Alain Colmerauer about 1970. Useful texts are Kowalski (1979a), Coelho, et al. (1980), and Clocksin and Mellish (1981). The discussion here is based mainly on PROLOG, with slight syntactic variants

* This reduction requires the inclusion of so-called Skolem functions, which take the role of existential quantification. These are not discussed here. Further discussion of clausal form is given in Nilsson, 1980, and Clocksin and Mellish, 1981.

to make it consistent with the preceding logical notation.

In logic programming, one typically distinguishes between *facts* and *rules*. A fact is a clause containing only the left hand side and no variables. For example,

MALE(DICK).

SIBLING(DICK, JANE).

are facts. Rules are clauses with expressions on both sides of the implication and containing variables. For example,

BROTHER(x, y) ← SIBLING(x, y) & MALE(x)

Disjunction is expressed using multiple rules. For example, BROTHER(x, y) can be proven in two ways, namely:

BROTHER(x, y) ← SIBLING(x, y) & MALE(x).

BROTHER(x, y) ← SIBLING(x, y) & MALE(y).

The first is the rule just discussed; the second allows for the reverse matching of arguments (because SIBLING is symmetric while BROTHER is not). Though this is the typical way of indicating disjunction in logic programming, for notational simplicity the connective, V, will sometimes be used. This is assumed to have lower priority than &. For instance,

BROTHER(x, y) ← SIBLING(x,y) & MALE(x) V SIBLING(y,x) & MALE(y).

is equivalent to:

BROTHER(x, y) ← (SIBLING(x,y) & MALE(x)) V (SIBLING(y,x) & MALE(y)).

Goal theorems (i.e. things to be proved) are denoted with a question mark, e.g.$,

BROTHER(DICK, JANE) ?

asks whether DICK is the brother of JANE. In this example the system would respond YES. Variables can also occur in goal theorems. In these cases the system's response is similar to that of database queries, namely, it returns all combinations of variable bindings that result in a provable theorem. For instance, the logic program:

MALE(DICK).

MALE(TOM).

MALE(HARRY).

MALE(x) ?

would respond:

x = DICK

x = TOM

x = HARRY

A slightly more complicated example is the following:

SIBLING(DICK, SALLY).

SIBLING(TOM, DICK).

SIBLING(HARRY, TOM).

SIBLING(x, z) ← SIBLING(x, y) & SIBLING(y, z).

The last rule indicates that the SIBLING relationship is transitive. Thus, the query,

SIBLING(x, SALLY) ?

results in the response:

x = DICK

x = TOM

x = HARRY

Note that three levels of inferencing are involved here. The first is simply a match to the fact, SIBLING(DICK, SALLY). The second requires the inference that TOM is a SIBLING to DICK and that DICK is a SIBLING to SALLY so TOM and SALLY must be SIBLINGs. The third is similar but with the additional inference that HARRY is SIBLING to TOM so that HARRY must be a SIBLING to DICK, hence also SIBLING to SALLY.

An important aspect of logic programming as compared with other types of computer languages is that it is non-procedural, or 'declarative'. In purely declarative languages, the order in which statements are evaluated is not controlled by the programmer*. Thus the order of the statements in a logic program doesn't matter as regards the system's inferencing capability. (It may however make a difference from an efficiency standpoint.) Logic programs are therefore an extreme form of modularity in computer program design.

However, there is one aspect of this non-procedurality that has to be compromised in order to address practical applications; this is for numeric computations. To do calculations in a strictly logical way would involve inferencing on the basic axioms of arithmetic. This would be

* This is true of 'pure' logic programming. In PROLOG, a certain amount of execution control can be specified by using the so-called 'cut' operator.

impossibly inefficient for any but trivial numeric computations. Logic programs therefore make calls to special subroutines when arithmetic is done. This is denoted here using a functional notation plus the usual arithmetic operators (+, -, *, /) for addition, subtraction, multiplication and division. For example, consider the following logic program:

HEIGHT-IN-METERS(DICK, 1.5).

HEIGHT-IN-FEET(x, z) ← HEIGHT-IN-METERS(x, y) & z = y * 3.28.

HEIGHT-IN-FEET(DICK, z) ?

z = 4.92.

Note that in logic programming, numeric constants are regarded as logical individuals. The subroutine invoked in computing z is logically regarded as a huge collection of facts giving all possible sums, products, etc.

What has just been described is the basic kernal of logic programming. Implementations include a variety of other aspects including in particular 'evaluable predicates' that have certain side effects permitting input/output, modification of assertions, etc. Also, more complex data structures (e.g. character strings, lists) are typically involved. These extensions enable logic programming to be used for a variety of applications beyond the usual conception of theorem-proving, e.g. natural language parsing, graph searches, user interfaces.

The motivation for introducing logic programming here is to examine the possibilities of database inferencing. This subject is considered next.

F. THE ENTITY-RELATIONSHIP INTERPRETATION

In the past decade, the Relational Model of Codd (1970) has clearly established the paradigm for database research. However, a criticism of the relational model is that it avoids commitment as to the semantics of the database, i.e. how the database structures signify or denote phenomena in the environment. A step in this direction is provided by the Entity-Relationship interpretation of Chen (1976). (This is normally called the Entity-Relationship *Model*, or ERM. It is, however, more an interpretation applied to the Relational Model.) The import of this approach is to draw attention to the role of relational keys. These are generally identifying labels for *entities* in the environment, e.g. part numbers, social security numbers. With this observation, certain relations serve to describe individual entities (entity relations), while others indicate relationships between entities (relationship relations).

The ERM highlights the *existential assumptions* of a database. Each tuple in a database is assumed to correspond to a particular entity in the environment or a relationship between entities.

The ERM is sometimes criticized that it fails to prescribe what count as entities, e.g. only physical objects? Should abstract objects also be admitted? The reply, of course, is that this depends on the organization's phenomenology. There is no absolute answer; what the organization recognizes as entities depends on its technology and view of the world. For instance, the popular example database

STUDENT (S#, ...)

COURSE (C#, ...)

ENROLLMENT (S#, C#, ...)

recognizes students and courses as entities, and enrollment as a relationship between them. This is a convenient view for university administrators, even though the concept of a 'course' is an abstraction that might be rather troublesome to pinpoint ontologically.

G. PREDICATE LOGIC INTERPRETATION

Ignoring, for the moment, the deeper semantic issues, the ERM has a straightforward interpretation in predicate logic:

a. entity relations = one-place predicates

b. relationship relations = multi-place predicates.

While this is a satisfactory interpretation of the definition of relations, the data in the relations are still unexplained. Generally, these seem to be of three types:

a. data items functioning as *identifiers* of entities (in the role of logical names)

b. data items corresponding to *predicates*.

c. data items representing *numeric measurements*.

For example, consider the relation:

EMPLOYEE	(NAME,	SEX,	SALARY)
	SMITH	MALE	35000
	JONES	FEMALE	42000

corresponding logical assertions would be:

EMPLOYEE(SMITH) & MALE(SMITH) & SALARY(SMITH, 35000).

EMPLOYEE(JONES) & FEMALE(JONES) & SALARY(JONES, 42000).

Here the values of the first attribute, NAME, translate as individual names in logic. The values of the second attribute, SEX, translate as predicate names, i.e. MALE(x), FEMALE(x). The values of the third attribute translate as numbers, which in logic programming are taken to be

another type of individual. To relate the human individual to the numeric individual, a two place predicate, SALARY(x, n), is introduced. Since the use of numbers in databases typically indicates a functional mapping from the real world entity to a numeric domain, a functional notation is often used, e.g.

SALARY(SMITH) = 35000.

SALARY(JONES) = 42000.

Database management models typically distinguish between the *structure* and *contents* of the database. In the logical form this distinction is not made. In database management, the structure/content distinction gives rise to the view of databases as repositories, somewhat akin to physical inventories. A database *query* specifies retrieval conditions, and the database contents that match these conditions are delivered to the user. In logical form queries are processed not simply by matching character strings, but rather by logical inference (this point is elaborated below).

This reflects a fundamental difference in the two perspectives. Database management regards data as character strings that the system stores and delivers to the user upon request. The *interpretation* of these character strings lies outside the theoretical concern. (Recall: GIGO = garbage-in-garbage-out; there is little in database management systems that requires that the data be meaningful.)

Representing data as logical assertions, however, one is more inclined to regard these as statements about the environment. This leads to a consideration of the epistemological evidence behind these asser-

tions, and the extrapolations and deductions that can be made from them.

A fundamental difference between logic programming and the more usual concept of theorem-proving is in the basic ontology. Theorem proving, following the usual pattern of logic, presumes some basic universe of discourse, e.g. numbers, blocks on a table. Logic programming, on the other hand, is much less restricted in this regard. In particular, much of logic programming is oriented towards objects that are data or syntactic structures. So, in addition to the more typical applications of predicate logic, logic programming may be used for example in sorting a list, or parsing natural language sentences. Thus, logic programming seems to blur the distinction between processing data structures and inferencing on logical assertions. For our purposes here, this ambiguity in logic programming serves as a useful bridge between the database management and logical views of databases.

H. RELATIONAL DATABASES AND LOGIC PROGRAMMING

Logic programming makes use of mechanical theorem-proving techniques as the basis for a general purpose programming language. The focus here is the use of logic programming for database inferencing.

The link between relational databases and logic programming is made by recognizing that, logically, a relation is the extension of a predicate. That is, a relation $P(x_1, ..., x_n)$ consists of all the n-tuples, $<x_1, ..., x_n>$, that satisfy the predicate, P. Thus, for example, the database:

EMPLOYEE	(NAME,	SEX,	SALARY)
	SMITH,	MALE,	35000
	JONES,	FEMALE,	42000

would be stated in a logic program as:

EMPLOYEE(SMITH, MALE, 35000).

EMPLOYEE(JONES, FEMALE, 42000).

Note that while the structure of the original relation is preserved, the attribute names are no longer used. Here the relation name, EMPLOYEE, is re-interpreted as a three place predicate and the attribute values in each tuple are its constant arguments. To refer to the entire relation, rather than individual tuples, attribute names might be translated as variables, e.g.,

EMPLOYEE(?NAME, ?SEX, ?SALARY).

However here, the former attribute names are merely arbitrary variable names. An equivalent designation would be:

EMPLOYEE(x, y, z).

Note how this example differs from its counterpart in the last section. Here EMPLOYEE is regarded as a single predicate, whereas before it was translated as a conjunct of three predicates. In conventional predicate logic the arguments of a predicate are normally regarded as names for individuals (in the universe of discourse). Here, on the other hand — and this is typical of most databases — not only do the arguments contain names for individuals, but other predicate names (e.g. MALE, FEMALE), as well as numbers (measurements).

As noted earlier, the ontology adopted by an organization (i.e., what basic individuals it recognizes) is a relative matter, depending on how it choses to view the world. (However, external reporting requirements may press it towards a more standardized ontology.)

In most databases, however, there is apparent recognition of an underlying ontology. This, again, is because data is generally retrieved in the same form that it is stored, without intermediate inferencing. For example, it is doubtful that any organization would regard FEMALE as naming a unique individual in the same sense that JONES does. But for database applications where logical inferencing is included, it becomes important to make this ontology explicit.

One of the simplest and perhaps most useful types of inferences for databases is for hierarchies of classification, so-called 'generalization hierarchies'. These were first proposed in the database

management literature by Smith and Smith (1977), though they were discussed in Artificial Intelligence some years earlier, e.g. Quillian (1968). An alternative notation, based on the Entity-Relationship interpretation, is given in Lee and Gerritsen (1978).

A generalization hierarchy is a graphical representation of a sequence of subset relationships between categories. An example of the Smiths' (1977:109) is reproduced in Figure [1].

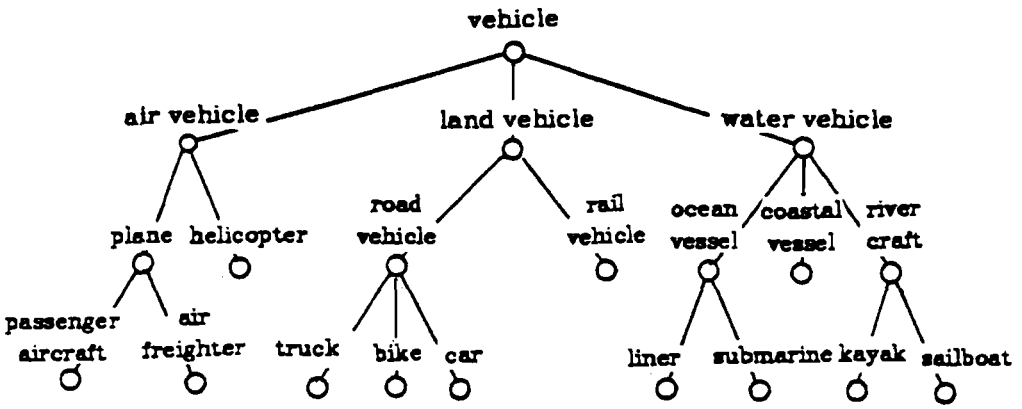

Figure [1]. A generic hierarchy over vehicles

The arcs in such generalization hierarchies are often read 'is a'. Thus an air vehicle 'is a' vehicle; a plane 'is a(n)' air vehicle; a passenger aircraft 'is a' plane; etc. Assuming that the primative predicates stored in the database are at the bottom of the tree, the generalization hierarchy translates into logic programming rules as follows:

PLANE(x) ← PASSENGER-AIRCRAFT(x).

PLANE(x) ← AIR-FREIGHTER(x).

AIR-VEHICLE(x) ← PLANE(x).

AIR-VEHICLE(x) ← HELICOPTER(x).

and so on. But now the ontological issues begin to emerge. Such inferences can only be made on relational attributes that are themselves predicates, and not, for instance, on attributes that are individual names or identifiers. In the terminology of the relational model, generalization hierarchies reflect the ambiguity that predicates may appear either as relation names or attribute values. These inferences are perfectly valid. However, they can only be recognized in the context of particular relations. Consider the following very simple example. Assume a relation:

EMPLOYEE(?ID, ?MARITAL-STATUS).

where ?ID is the employee's identification code and ?MARITAL-STATUS can have the values SINGLE, MARRIED or DIVORCED. (Note that the above expression is not a complete logic programming statement, as it is neither a fact or a rule. This expression could however be entered as a query, with a "?" following, which would then return all the tuples of the relation.) To plan office parties, we would like to specify:

ELIGIBLE(x) ← SINGLE(x) V DIVORCED(x).

However, having marital status as an argument of the employee relation, we are led to define it as follows:

EMPLOYEE2(?ID, ELIGIBLE) ← EMPLOYEE(?ID, SINGLE) V EMPLOYEE(?ID, DIVORCED).

The important thing to note is that we are forced to create a new relation name, EMPLOYEE2. The difference between EMPLOYEE and EMPLOYEE2 is that the latter has a different interpretation of its second argument.

The difficulty is that in typical relational form, with features (predicates) appearing as arguments, the governing predicate name carries the sense of these features implicitly. In the above example, the term EMPLOYEE carried not only the sense of employment, but also assertions about marital status.

Further deductive rules would entail the invention of further variants of the employee relation, e.g. EMPLOYEE3, EMPLOYEE4, each having its own peculiar interpretation of arguments. Hence, as the deductive rules become more complex, it becomes advantageous from the standpoint of conceptual clarity to promote these embedded features to the status of explicit predicates. Continuing the previous example, we would have:

EMPLOYEE(x) ← EMPLOYEE(x,y).

(Note: like-named predicates with different numbers of arguments are regarded as different predicates.)

SINGLE(x) ← EMPLOYEE(x, SINGLE).

MARRIED(x) ← EMPLOYEE(x, MARRIED).

DIVORCED(x) ← EMPLOYEE(x, DIVORCED).

ELIGIBLE(x) ← SINGLE(x) V DIVORCED(x).

Another type of data typically appearing in databases is numeric measurement. A similar rationale applies. As inferencing on the features indicated by these measurements becomes more complex, it becomes advantageous to separate out these features explicitly. For example,

consider the relation:

BUILDING(?ADDRESS, ?HEIGHT-IN-METERS).

To convert to feet, we would like to specify the rule:

HEIGHT-IN-FEET(x,n) ← HEIGHT-IN-METERS(x,m), & n = m * 3.28.

However, as embedded in these relations, separate rules for the units conversion would be needed for each length attribute of each relation, e.g.,

BUILDING2(x,z) ← BUILDING(x,y) & z = y * 3.28.

Again we are faced with the introduction of the confusing terminology BUILDING2. Like before, the problem stems from the interpretation of the predicate name BUILDING to include more than the elementary concept of buildinghood, but also the measurement of that building's height. To distinguish these concepts explicitly, we would use the rules:

BUILDING(x) ← BUILDING(x,h).

HEIGHT-METERS(x,h) ← BUILDING(x,h).

(*) HEIGHT-FEET(x,z) ← HEIGHT-METERS(x,y), z = y * 3.28.

Having distinguished 'height' explicitly, we can now make use of this unit of measure conversion for other entities having the feature of height. For example, another relation might be:

PERSON(?ID, ?HEIGHT-IN-METERS)

To separate the concept 'person' from his or her height measurement, we add the rules:

PERSON(?ID) ← PERSON(?ID, ?H).

HEIGHT-METERS(x,y) ← BUILDING(x,y).

By using the rule (*) above, we may now infer the height in feet of any building recorded in the database.

Likewise, with the concepts 'building' and 'person' separately distinguished, we may want to add additional deductive rules about them. For instance,

PHYSICAL-OBJECT(x) ← PERSON(x) V BUILDING(x).

i.e. persons and buildings are both physical objects. With this abstraction, general knowledge pertaining to physical objects can then be added, e.g. that they have mass, height.

These examples reflect an important insight suggested by the graphical notation of generalization hierarchies. One would like to specify deductive rules to apply as generally as possible. For instance, it is a characteristic of vehicles of all types that they may change from one location to another. It is a characteristic of all water vehicles that their location will always be in some body of water. In normal database representations, one would have to specify these inferences repeatedly for each of 'submarine', 'kayak', 'sailboat', etc.

I. CONCLUDING REMARKS

As was shown, logic programming can be applied directly to relational databases (conceptually speaking, ignoring implementation considerations) to transform one relational view into another. However, this does not fully exploit the capabilities of logical inferencing. Greater flexibility is achieved as relations are decomposed into more elementary predicates. This requires a recognition of the basic ontology of the database, i.e. the elementary types of individual entities in the environment that are recognized. Predicates should ideally have only names for these elementary entities as arguments.

However, to avoid the computational difficulties involved with logically deducing arithmetic operations, a compromise was suggested, admitting numbers into the basic ontology. These are separated syntactically, as measurement functions mapping real world individuals to numeric individuals.

Various implementation issues arise from the above proposal. For instance, it is implicitly presumed that the organization maintains a conventional relational database for data processing purposes, and that these inferencing structures are built 'on top'. The mechanisms for this interface have been ignored in this discussion.

Beyond this, however, logical inferencing on databases raises another set of issues that are not so much computational as epistemological. The ontological aspects have been touched on briefly. Other aspects include cross-temporal relationships as well as the special problems involved with non-factual contexts, e.g. predictions, plans and contractual commit-

ment. These aspects are further disussed in Lee (1980, 1981, 1983, 1983a).

REFERENCES

Bonczek, R.H., C.W. Holsapple and A.B. Whinston. 1981. *Foundations of Decision Support Systems*. New York: Academic Press.

Chen, P.P-S. 1976. The Entity-Relationship Model —Toward a Unified View of Data. *ACM Transactions on Database Systems* 1(March):9-36.

Clocksin, W.F. and Mellish, C.S. 1981. *Programming in Prolog*. New York: Springer-Verlag.

Codd, E.F. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(June):377-387.

Codd, E.F. 1972. Further Normalization of the Data Base Relational Model. In R. Rustin, ed., *Data Base Systems* Courant Computer Science Symposia 6. Englewood Cliffs, New Jersey: Prentice-Hall., also IBM Research Report RJ909.

Codd, E.F. 1979. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 44(December):397-434.

Coelho, H., Cotta, J.C. and Pereira, L.M. 1980. *How to Solve It With Prolog*. 2nd Edition. Lisbon: Laboratório Nacional de Engenharia Civil.

Date, C.J. 1977. *An Introduction to Database Systems*, 2nd edition. Reading, Massachusetts: Addison-Wesley.

Davis, R., and J. King. 1975. An Overview of Production Systems, Stanford

AI Lab Memo AIM-271, *Stanford Computer Science Report*. STAN-CS-75-524, October.

Davis, R., and Lenat, D.B. 1982. *Knowledge- Based Systems in Artificial Intelligence*. New York: McGraw-Hill International.

Fagin, R. 1977. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems, 2(3):262- 278.*

Fick, G. and Sprague, R.H. Jr., eds. 1980. Decision Support Systems: Issues and Challenges. Oxford: Pergamon Press.

Gallaire, H., Minker, J. and Nicolas, J.M., eds. 1981. *Advances in Data Base Theory* Volume 1. New York and London: Plenum Press.

Gallaire, H., and Minker, J., eds. 1978. *Logic and Data Bases*. New York and London: Plenum Press.

Gorry, G.A. and Scott-Mortonm M.S.. 1971. A Framework for Management Information Systems. *Sloan Management Review*, 13(1):55-70.

Infotech. 1981. *Machine Intelligence*. Infotech State of the Art Report,

Keen, P.G.W. and Scott-Morton, M.S. 1978. *Decision Support Systems*. Reading, Massachusetts: Addison-Wesley.

Kent, W. 1978. *Data and Reality*. Amsterdam: North-Holland.

Kent, W. 1979. Limitations of Record-Based Information Models. *ACM Transactions on Database Systems*, 4(1):107-131.

Kowalski, R. 1979. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424-436.

Kowalski, R. 1979a. *Logic for Problem Solving*. New York and Oxford: North Holland.

Lee, R.M., and Gerritsen, R. 1978. Extended Semantics for Generalization Hierarchies. *Proceedings of the International Conference on management of Data*. Austin, Texas, June, 1978, pp.18-25.

Lee, R.M. 1980. CANDID: A Logical Calculus for Describing Financial Contracts. Ph.D. dissertation, abailable as WP-80-06-02. Philadelphia, PA: Department of Decision Sciences, the Wharton School, University of Pennsylvania.

Lee, R.M. 1981. CANDID Description of Commercial and Financial Concepts: A Formal Semantics Approach to Knowledge Representation. WP-81-162. Laxenburg, Austria: International Institute for Applied Systems Analysis.

Lee, R.M. 1983. Epistemological Aspects of Knowledge-Based Decision Support Systems. In H.G. Sol, ed., *Processes and Tools for Decision Support*, Amsterdam: North-Holland.

Lee, R.M. 1983a. *Data and Language in Organizations: Epistemological Aspects of Management Support Systems.* London: Academic Press, to appear.

McDermott, D. 1980. The Prolog Phenomenon. *SIGART Newsletter,* 72(July):16-20.

Newell, A., J. Shaw, and H. Simon. 1963. Empirical Explorations of the Logical Theory Machine. In *Computers and Thought,* E. Feigenbaum and J. Feldman (eds.), New York: McGraw-Hill, pp.109-113.

Nilsson, N.J. 1980. *Principles of Artificial Intelligence,* Palo Alto, CA: Tioga Publishing Co.

Robinson, R.A. 1965. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM,* 12(January):23-41.

Smith, J.M. and Smith, D.C.P. 1977. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems,* (2) June: 105-133.

Sol, H.G. 1983. *Processes an Tools for Decision Support.* Proceedings of IFIP/IIASA Working Conference on Processes and Tools for Decision Support, July 19-21, 1982, Laxenburg, Austria. Amsterdam: North-Holland.

Stamper, R. 1973. *Information in Business and Administrative Systems.* New York: Wiley.

Tsichritzis, D. and Klug, A. eds. 1977. The ANSI/X3/SPARC DBMS Framework. Report of the Study Group on Data Base Management Systems. Montvale, New Jersey: AFIPS Press.