DATA BASES: A LOGICAL PERSPECTIVE

Raymond Reiter
Department of Computer Science
University of British Columbia
Vancouver, B.C.
Canada

My work in data base theory is a natural outgrowth of my longstanding concern with the problem of representing and reasoning with domain specific knowledge, a problem of major concern in Artificial Intelligence. In data base terminology this is the conceptual modelling issue. My own methodological bias favours logic as a representation language for conceptual modelling, a bias which historically arose within AI in response to AI's emphasis on the ability to reason deductively with representations. In this position paper I shall argue that logic has other advantages for data base theory. Specifically my objective is to provide the outline of a logical reconstruction of certain aspects of conventional data base theory.

The Language of First Order Logic

In order to fix the discussion I shall focus upon first order logic, although for some applications a weak second order logic is required. So, begin with a suitable alphabet of

Variables: $x,y,z,x_1,y_1,z_1,\ldots,$

Constants: John-Doe, Acme, part33, ...,

Predicates: SUPPLIES(•,•), EMPLOYEE(•), ...,

Logical Constants: $\supset$ (implies), $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\equiv$ (if and only if)

Quantifiers: (x) (for all x), (Ex) (there exists an x).

Using this alphabet, well formed formulae may be constructed in the usual way. For example:

SUPPLIES(Acme,part33): Acme supplies part33.

EMPLOYEE(John-Doe); John-Doe is an employee.

$(x)[\text{EMPLOYEE}(x) \supset \text{HUMAN}(x)]$: All employees are human

$(x)(y)(z)[\text{SUPPLIES}(x,y) \wedge \text{SUBPART}(z,y) \supset \text{SUPPLIES}(x,z)]$;

All parts suppliers supply all subparts for those parts.

The language of first order logic has a well known denotational semantics, due originally to Alfred Tarski. I omit the details.

The Concept of a Data Base

Define a data base to be any finite set of formulae. This is clearly too broad a definition for practical purposes. Which leads us to

Data Models

Data models specify structure for data bases. They may do so by restricting the allowable formulae in the data base, by partitioning the data base, or by imposing various computational requirements on it.[1] Some examples:

1. The introduction of types [McSkimin and Minker 1977, Reiter 1977, 1980].

A type is a distinguished unary predicate, e.g. EMPLOYEE(•), SUPPLIER(•). That's all. Notice that I am not identifying a type with the set of its instances in the data base, or with the set of all of its possible instances, whatever that might mean.

Having fixed upon some set of types for an application, one can then easily represent their standard properties:

[1] I was criticized at the workshop for not providing a formal definition of "data model". Obviously I still can't; nor do I believe that this is possible in any general way. Mea culpa.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1980 ACM 0-89791-031-1/80/0600-0174 $00.75

Subtypes: $(x)[EMPLOYEE(x) \supset HUMAN(x)]$; EMPLOYEE is a subtype of HUMAN

Disjointness: $(x)\neg[MALE(x) \wedge FEMALE(x)]$; MALE and FEMALE are disjoint types.

New types may be defined by arbitrary Boolean operations on primitive types:

$(x)[BACHELOR(x) \equiv MALE(x) \wedge SINGLE(x)]$

The introduction of types naturally induces a partition of a data base into two disjoint components: a type data base of formulae whose only predicates are types, and a principal data base consisting of the remaining formulae.

2. The concept of a relation.

Assume that a type structure has been defined for a data base DB. Assume further that for each non type predicate R of DB, DB contains a formula of the form:

$(x_1)\ldots(x_n)[R(x_1,\ldots,x_n) \supset D_1(x_1) \wedge \ldots \wedge D_n(x_n)]$

where $D_1,\ldots,D_n$ are types. Then R is a relation and $D_1,\ldots,D_n$ are its domains. If R is an n-ary relation and $c_1,\ldots,c_n$ are constants, then $(c_1,\ldots,c_n)$ is an instance of R (relative to DB) iff $DB \vdash R(c_1,\ldots,c_n)$.[1] The extension of a relation (relative to DB) is the set of its instances.

3. Decidability of the type membership problem.

Given the uses for which they were intended, it is natural to require that types be computable, as follows:

For all types τ and constants c, either $TDB \vdash \tau(c)$ or $TDB \vdash \neg\tau(c)$ where TDB is the type data base. In other words, the type membership problem is decidable.

4. The domain closure axiom [Reiter 1980].

Normally, data base systems make the implicit assumption that, for any given state of the data base, the individuals explicitly mentioned in that data base are the only existing individuals. This assumption is manifest in the way such systems treat universal quantifiers in the evaluation of queries. Logically this assumption can be made explicit by including, in the data base, the following domain closure axiom:

$(x)[x=c_1 \vee \ldots \vee x=c_n]$

where $c_1,\ldots,c_n$ are all of the known individuals.

[1] If W is a set of first order formulae, and w is a first order formula, then $W \vdash w$ means that there is a proof of w from premises W.

5. The closed world assumption [Reiter 1978a]

There is an interesting logical structure underlying the treatment of negation by current data base systems. Typically such systems will conclude of a relation R and tuple $(c_1,\ldots,c_n)$ of constants that $\neg R(c_1,\ldots,c_n)$ is the case whenever $R(c_1,\ldots,c_n)$ is absent from the data base.[1] This treatment of negation generalizes as the following closed world assumption:

If $DB \nvdash R(c_1,\ldots,c_n)$ then $\neg R(c_1,\ldots,c_n)$ "holds"[2], where $\nvdash$ means not provable.

First Order Query Languages

Since a data base is a set of first order formulae, it is natural to interrogate a data base using some sort of first order query language. I shall describe a simplified version of the one used in [Reiter 1978b]. A query is any expression of the form

$\langle x_1,\ldots,x_m | (q_1y_1)\ldots(q_ny_n)W(x_1,\ldots,x_m,y_1,\ldots,y_n)\rangle$

where $W(x_1,\ldots,x_m,y_1,\ldots,y_n)$ is any quantifier free first order formula with free variables $x_1,\ldots,x_m$, $y_1,\ldots,y_n$, and (q_iy_i) is either the universal quantifier (y_i) or the existential quantifier (Ey_i). Intuitively such a query denotes the set of all tuples $(x_1,\ldots,x_m)$ such that $(q_1y_1)\ldots(q_ny_n)W(x_1,\ldots,x_m,y_1,\ldots,y_n)$ is true in a given data base.

For example:

$\langle x | (Ey)[SUPPLIER(x) \wedge PART(y) \wedge SUPPLIES(x,y)]\rangle$; all suppliers who supply parts.

$\langle x | (Ey)(z)[TEACHER(x) \wedge CALCULUS(y) \wedge TEACH(x,y) \wedge \{ALGEBRA(z) \supset \neg TEACH(x,z)\}]\rangle$; all teachers who teach calculus but not algebra.

Formally, define a tuple $(c_1,\ldots,c_m)$ of constants to be an answer to a query (with respect to a given data base DB) iff

$DB \vdash (q_1y_1)\ldots(q_ny_n)W(c_1,\ldots,c_m,y_1,\ldots,y_n)$[3]

The value of a query is then the set of all of its answers. [Reiter 1978b] describes a method for

[1] I am adopting the point of view that the data in a relational data base is simply a set of ground atomic formulae i.e. formulae of the form $R(c_1,\ldots,c_n)$.

[2] The scare quotes indicate that the notion of a formula "holding" requires some elaboration. See [Reiter 1978a].

[3] The notion of an answer to a query is actually somewhat more complicated than this. See [Reiter 1978b] for the minutiae.

computing the value of an arbitrary query for a certain large class of data bases. See also [Minker 1978].

Data Base Integrity

Any first order formula may be distinguished as an integrity constraint. I shall henceforth assume that a data base is composed of two disjoint components: DB, a data base in the old sense, and IC, a finite set of integrity constraints. The point of view which I want to formalize is that IC is a set of invariant properties which DB must satisfy after each transaction. Now an invariant of any mathematical system is simply a property which provably must hold for that system. This motivates the following definition:

DB satisfies IC=$\{I_1,\ldots,I_n\}$ iff DB $\vdash I_1 \wedge \ldots \wedge I_n$.

Notice that if DB is inconsistent then it automatically satisfies any set of integrity constraints. So there are really two problems here: One is to ensure that DB is consistent after a transaction; the other is to verify that it satisfies the integrity constraints.

Notice that this definition provides a foundation for the proof of appropriate meta-theorems relative to a given application. An "appropriate meta-theorem" is something of the form:

If T is a transaction satisfying certain properties and if, prior to T, DB is consistent and satisfies IC, then after T DB is also consistent and satisfies IC.

A different point of view regarding the logical status of integrity constraints is described in [Nicolas and Yazdanian 1978].

Discussion

I have attempted to provide the foundations for a purely logical treatment of data base theory.[1] There are a number of advantages to such an approach:

1. Logic has great expressive power and provides a natural representation language for conceptual modelling.
2. It has a well defined denotational semantics.
3. It has a well developed proof theory which thus provides a basis for inferential retrieval of answers, and for the detection of integrity violations.
4. Just as the relational calculus provides a reference standard against which the power of relational query languages may be compared, so does logic provide a standard for evaluating the inferencing power of alternate modelling formalisms.
5. Logic can serve as a unifying representation language for conceptual modelling; different non logical modelling formalisms may be compared and evaluated by first translating them into logical terms.
6. In the same vein as 5, logic can be used to specify the semantics of non logical representation notations. For example, the meanings of all of the graphical notations that I know of for conceptual modelling can be completely and unambiguously specified by a translation into logic.

[1] Space limitations have prevented my discussing the logical status of a number of issues of concern to data base theory, for example virtual relations and views. For the same reason I have in no way done justice to the literature on logic and data bases. Notable ommisions are various papers by C.L. Chang, H. Gallaire, C. Green, C.H. Kellogg, R. Kowalski, J. Minker and J.M. Nicolas.

References

Minker, J. (1978). An experimental relational data base system, in Logic and Data Bases, H. Gallaire and J. Minker (eds.), Plenum Press, N.Y.

McSkimin, J.R. and Minker, J. (1977). The use of a semantic network in a deductive question-answering system, PROC. IJCAI-77, Cambridge, Mass., 50-58.

Nicolas, J.M. and Yazdanian, K. (1978). Integrity checking in deductive data bases, op. cit., [Minker 1978].

Reiter, R. (1977). An approach to deductive question-answering, Bolt Beranek and Newman Inc., Cambridge, Mass., Tech. Report 3649, 161 pp.

_____ (1978a). On closed world data bases, op. cit., [Minker 1978].

_____ (1978b). Deductive question-answering on relational data bases, ibid.

_____ (1980). Equality and domain closure in first order data bases, J.ACM, 27, 2, 235-249.