Chapter IV. Data Bases as Value Specifications.

1. Introduction.

In real-world, non-experimental computer programs which answer queries about the state of affairs in a subject domain, the state of affairs is normally represented by a data base – a formatted collection of data stored on magnetic disk or in another form of mass memory.

Many question answering systems (including PHLIQA1) are, in fact, "natural-language data base interfaces" which are meant to function as the front end of a query evaluation system which employs an existing data base management system. Therefore, a satisfying account of operations on data bases is an important component of a theory of computational question answering.

In the treatment of questions and answers developed in Chapter II, both questions and answers were represented as expressions in a logical language and the connection between such expressions and states of affairs in the subject domain was constructed through logical model theory. To enable an account of the way in which a data base may be used to answer questions, the meaning of a data base must be described in terms of the same basic semantic notions that were used earlier to account for the meaning of questions and answers. We must therefore establish a connection between the content of a data base and the possible interpretations of a logical language.

The main thesis of this chapter is that this connection is, in fact, quite simple. We shall argue that any data base schema should be considered as specifying the descriptive constants of a logical language, while any particular data base within a schema specifies an interpretation of that logical language. The notion of a "value specification" – a formal object which induces an interpretation on a logical query language – will figure prominently in the discussion. This perspective is being advanced as an alternative to the rather widespread idea that data bases should be analyzed as collections of first order axioms. (See section 6, below). An important argument in its favour is that it accounts in a very direct way for the correctness of recursive query evaluation procedures.

The viability of the perspective on data bases advocated here has been pointed out before. (Scha, 1977; Nicolas and Gallaire, 1978; Bronnenberg et al., 1980; Konolige, 1981. Important connections with Codd's work (1970) will be considered specifically in section 4 below). However, no comprehensive formal articulation of this way of conceptualizing data bases has been presented previously.
2. **Value Specifications.**

We have argued earlier that a subject domain may be characterized by a definition of the descriptive constants of a logical language and their intuitive meanings. The state of affairs in a subject domain could then be characterized by an *interpretation* of this language, which specifies the denotation of every descriptive constant. In the present chapter, we shall show that specifying a formal object which indicates the denotation of every constant in the query language – i.e. the extension of the interpretation function – is a way of specifying the state of affairs in the subject domain which ties in directly with the definition of the semantics of the query language.

To be able to construct such a formal object, we assume that there is a "value language" corresponding to the query language, which has a "logical proper name" for every individual in the domain. This language has exactly the same syntactic constructions and the same semantics as the query language as well as exactly the same type system, but it contains no descriptive constants. The value language is syntactically defined as a logical language with only "individual" constants (i.e. the type of every constant is an atomic type). The semantics of a value language is defined in the usual way with two extra constraints on the interpretations of the language:

1. Every individual constant denotes a distinct entity
2. Every entity in the domain of a descriptive atomic type is denoted by an individual constant.

A *value specification* may now be defined as a collection of pairs \(<c,e>\) which specifies the extension of a function which assigns to every descriptive constant \(c\) of the logical language an expression \(e\) of the value language. Since the value language expressions have the same denotation for every interpretation of that language, this function induces the one interpretation

---

1) This way of adapting the definition of interpretations is not as ad hoc or far-fetched as it may seem. Logical systems in which individual constants were given this special role have a venerable history (e.g. Wittgenstein (1922), Carnap (1947) and Reiter (1977)). A difficulty with this approach is that the language fixes the number of individuals in the domain, although for data bases about a fragment of the real world, the individuals in the domain are not usually fixed. (Pott, 1976). There are at least three alternative ways of dealing with this problem:

1. Adopt a "loose ontology" by assuming a fixed, infinitely large domain of *possible individuals*. The set of *actual individuals* is then defined as the union of the domains of the atomic types, constituting a subset of the possible individuals.
2. Abstain from defining one particular value language. Instead define the set of all possible value-languages where these languages differ from each other in having different sets of individual constants.
3. Formulate the queries so that they do not refer to real world entities. (Questions which ask about real world entities may be transformed into questions which ask about their names and other "formal" properties (Chapter II, section 7). The technique of "identification-translations" may be used to identify real world entities in terms of their names (Chapter V, section 4).)


on the logical language which assigns to every descriptive constant the denotation of the expression assigned to it by the value specification.

There is an extremely simple algorithm which, given a value-specification, could be used to turn any query expression into an expression which satisfies many of the requirements of an adequate answer-expression. Simply by substituting values for constants according to the value specification, an L-determinate expression is generated which has the same denotation as the query expression. However, this expression is, in general, unnecessarily complicated. Instead, a recursive evaluation algorithm is therefore used to turn a query expression into a canonical value expression which can be used as an answer.\(^2\) Exactly how such an algorithm functions will be discussed after describing the nature of canonical value-expressions.

**Canonical value-expressions**

For every finite denotation of the value language expressions, we define one canonical value expression with that denotation.

If a data base implements a value-specification, a query-expression may be evaluated by means of a collection of recursive procedures which correspond closely to the recursive definition of the semantics of the language. These procedures operate on canonical representations of the values of various types.

The canonical representations of values may be defined as the expressions of a canonical value-language:

1. Every individual constant is a canonical value-expression.
2. If \(A_1, \ldots, A_n\) are canonical value-expressions of types \(\alpha_1, \ldots, \alpha_n\), then:
   - \(\text{set}(\langle A_1, \ldots, A_n \rangle)\) is a canonical value-expression of type \(S(\cup (\alpha_1, \ldots, \alpha_n))\),
   - \(\text{bag}(\langle A_1, \ldots, A_n \rangle)\) is a canonical value-expression of type \(B(\cup (\alpha_1, \ldots, \alpha_n))\),
   - \(\text{list}(\langle A_1, \ldots, A_n \rangle)\) is a canonical value-expression of type \(L(\cup (\alpha_1, \ldots, \alpha_n))\),
   - \(\text{file}(\langle A_1, \ldots, A_n \rangle)\) is a canonical value-expression of type \(F(\cup (\alpha_1, \ldots, \alpha_n))\).
3. If \(A_1, \ldots, A_n\) are canonical value-expressions of type \(\alpha_1, \ldots, \alpha_n\), then \(\langle A_1, \ldots, A_n \rangle\) is a canonical value-expression of type \(\langle \alpha_1, \ldots, \alpha_n \rangle\).
4. If \(A_1, B_1, \ldots, A_n, B_n\) are value-expressions of type \(\langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_n, \beta_n \rangle\), resp., then \(\text{function} (\langle A_1, B_1 \rangle, \ldots, \langle A_n, B_n \rangle)\) is a value-expression of type \((\alpha_1, \ldots, \alpha_n) \rightarrow (\beta_1, \ldots, \beta_n)\).
5. If \(N\) is a canonical value-expression of type integer or real and \(E\) is a canonical value-expression of type \(\epsilon\), then \((\text{num: } N, \text{ unit: } E)\) is a canonical value-expression of type AMT (\(\epsilon\)).

\(^2\) This is also the case in PHLIQA1. In Bronnenberg et al. (1980) the PHLIQA1 program was described misleadingly as using the substitution algorithm.
6. If $A$ is a canonical value-expression of type $\alpha$ then, for any integer $i$, $\text{id}_i(A)$ is a canonical value-expression of type $\text{ID}_i(\alpha)$.

A normalized value specification assigns to every descriptive constant of a logical language a canonical expression of the corresponding value language. Canonical values may be used, therefore, to formalize the definition of the semantics of a logical language by correlating every syntactic rule which forms an expression out of sub-expressions with a semantic rule which defines the denotation of the expression in terms of the denotations of the sub-expressions. $^3$ Together, these semantic rules define the denotation of any expression in terms of the denotation of the constants occurring in it. If canonical values have been defined, we may define formal correlates of the semantic rules for the case of finite denotations as functions which define the value of an expression in terms of the value of its sub-expressions. $^4$ These functions, taken together, define the value of any expression in terms of the values of its constants.

Given a formalization of the semantic definition of a logical language, we may construct an algorithmic definition. Every function which defines the value of a certain kind of expression in terms of the values of its sub-expressions, may be implemented by an effective procedure which computes the value of the expression on the basis of the values of its subexpressions. Together, these procedures constitute an effective recursive algorithm for computing the value of an expression on the basis of the values of the constants occurring in it. Thus, a value specification of the normalized variety makes it possible to compute the answer to a query which is formulated in the appropriate logical language, by means of a simple recursive evaluation procedure.

More refined versions can be easily imagined. For instance, the PHLIQA1 procedures which compute the values of quantification-, selection- and iteration-expressions do not always evaluate both their sub-expressions. They have two sub-expressions, one denoting a set, the other denoting a function. If the function-expression is a lambda-expression, its whole extension is not computed. Instead, the value of the set-expression is used as the range of the lambda-variable, and only the values of the relevant instances of the body of the lambda-expression are computed.

It is clear that value-specifications may be implemented as data bases of some sort. The descriptive constants of the logical language may then be

$^3$ For the sake of simplicity of expression, the notion "denotation of sub-expressions" is taken to encompass the domains of the types of (binding occurrences of) variables.

$^4$ This means that we must restrict our attention to expressions not containing formal constants with infinite denotations. Variables ranging over infinite formal domains are thus also excluded.
viewed as defining data base schemata. All value-specifications for these constants may be stored as homogeneous collections of data according to a format which is determined by the type of the constant. Conversely, the most important data models underlying existing data base management systems can be seen as implementations of value-specifications. In the next sections, we shall show how this is the case in the Relational Model and the CODASYL Model, and argue that other "abstract data models" can be formalized in this way as well.

4. Relational Data Bases viewed as Value Specifications.

Codd (1970) defined relational data bases as collections of tables, where a table is a set of n-tuples of individuals. The elements within every n-tuple may be identified by the integers 1 … n or by n names called attributes. An n-tuple within a table is identified by the keys of the table, a subset of the attributes. The relational terminology and query languages like Relational Algebra suggest that data bases be viewed as specifying the values of constants denoting sets of n-tuples. The fact that certain elements identify the n-tuple in which they occur, however, leads to a different perspective.

Instead of saying that a table specifies the extension of an n-ary relation, we say that it specifies the extension of a partial function from k-tuples to n-k-tuples, if k elements of a tuple constitute the primary key together. This means that a table is read as a translation rule of the form

\[ F \Rightarrow \text{function} (\langle \text{tuple}_2 (\text{tuple}_k (B_{11}, \ldots, B_{1k}), \text{tuple}_{n-k} (B_{1k+1}, \ldots, B_{1n})), \text{tuple}_2 (\text{tuple}_k (B_{m1}, \ldots, B_{mk}), \text{tuple}_{n-k} (B_{m k+1}, \ldots, B_{mn}))) \]

where \( F \) is a query-language constant of type \( \langle \alpha_1, \ldots, \alpha_k \rangle \rightarrow \langle \alpha_{k+1}, \ldots, \alpha_n \rangle \)

The types \( \alpha_1, \ldots, \alpha_n \) are, in relational terminology, the domains of the relation A: they indicate the set of values that may occur in the corresponding "slot" in the n-tuple. This domain must either be specified in the data base (by occurring as the domain of a single key of a relation) or it must be known independently of the data base (in our terminology: it is a formal type, such as integer or string).

Consider, for example, the relation S in Date (1975). This table represents information about suppliers, identified by a code which is the value of the attribute S#. The table gives the name, the status and the city of every supplier:
<table>
<thead>
<tr>
<th>S#</th>
<th>NAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
</tr>
</tbody>
</table>

The table can be read as the translation rule

\[ F_S \implies \text{function (} \langle \langle S1, \langle "Smith", 20, "London" \rangle \rangle \right) \]

\[ \langle S2, \langle "Jones", 10, "Paris" \rangle \rangle \right) \]

\[ \langle S3, \langle "Blake", 30, "Paris" \rangle \rangle \right) \]

\[ \langle S4, \langle "Clark", 20, "London" \rangle \rangle \right) \]

\[ \langle S5, \langle "Adams", 30, "Athens" \rangle \rangle \right) \)

We are arguing that this view of relational data bases has advantages over the more strictly relational view which would consider a table in a data base as a translation rule of the form

\[ A \implies \text{set (} \langle \text{tuple}_n (B_{11}, ..., B_{1n}),} \]

\[ \text{tuple}_n (B_{m1}, ..., B_{mn}) \rangle \),

where \( A \) is a query-language constant of type \( S (\langle \alpha_1, ..., \alpha_n \rangle) \).

The relation \( S \) above would then be analyzed as

\[ S \implies \text{set (} \langle \text{tuple}_4 (S_1, "Smith", 20, "London")), \]

\[ \text{tuple}_4 (S_2, "Jones", 10, "Paris"), \]

\[ \text{tuple}_4 (S_3, "Blake", 30, "Paris"), \]

\[ \text{tuple}_4 (S_4, "Clark", 20, "London"), \]

\[ \text{tuple}_4 (S_5, "Adams", 30, "Athens")) \rangle \)

If the data base is analyzed as (5), there would be a problem justifying that a formula such as

\[ \text{tuple}_4 ("S1", "Jones", 30, "London") \in S \]

can be evaluated as \text{FALSE} on the basis of (2) by only inspecting the first entry. What would be needed to account for this would be an axiom like
\[ \forall u, x_1, x_2, y_1, y_2, z_1, z_2: ((\text{tuple}_4 (u, x_1, y_1, z_1) \in S \land \text{tuple}_4 (u, x_2, y_2, z_2) \in S) \Rightarrow (x_1 = x_2 \land y_1 = y_2 \land z_1 = z_2)) \]

Evaluation procedures actually used in data bases, however, do not make use of such axioms. Their operations can be accounted for by an analysis such as (3). In terms of that perspective on data bases, (6) would be formulated as

\[ F_S (<S_i>) = <"Jones", 30, "London"> \] (7)

By consulting value-specification (3), this can be equated to

\[ <"Smith", 20, "London"> = <"Jones", 30, "London"> \] (8)

and further to

FALSE (9)

As another example of the data base analysis we propose, consider the relation SP from Date (1975), represented by the following table:

<table>
<thead>
<tr>
<th>SP</th>
<th>S#</th>
<th>P#</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>P1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>P2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>P3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>P4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>P5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>P6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>P1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>P2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>P3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>P5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>P2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>P4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>P5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>P5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

(10)

The combination of the values of the attributes S# (supplier-identification) and P# (part-identification) identifies a row in this table. The table can be read as representing the following rule:
\[
F_{SP} \implies \text{function (}
\langle
\langle S1, P2, \langle 3 \rangle \rangle,
\langle S1, P2, \langle 2 \rangle \rangle,
\langle S1, P3, \langle 4 \rangle \rangle,
\langle S1, P4, \langle 2 \rangle \rangle,
\langle S1, P5, \langle 1 \rangle \rangle,
\langle S1, P6, \langle 1 \rangle \rangle,
\langle S2, P1, \langle 3 \rangle \rangle,
\langle S2, P2, \langle 4 \rangle \rangle,
\langle S3, P3, \langle 4 \rangle \rangle,
\langle S3, P5, \langle 2 \rangle \rangle,
\langle S4, P2, \langle 2 \rangle \rangle,
\langle S4, P4, \langle 3 \rangle \rangle,
\langle S4, P5, \langle 4 \rangle \rangle,
\langle S5, P5, \langle 5 \rangle \rangle
\rangle)
\]

(11)

5. CODASYL Data Bases viewed as Value Specifications.

A CODASYL system (CODASYL DBTG, 1971) can be used to implement many different kinds of abstract data models (e.g. relational data bases – see Lacroix (1977)). But a CODASYL data description may also be considered as an abstract data model in its own right. It is sufficiently well-structured for that, and the efficiency of an implementation may be enhanced if there is a direct correspondence between the query language and the data base as implemented, without intermediate models. We shall now discuss how the constants of a query-language may be derived from a CODASYL data base which only uses the most important CODASYL concepts.

Described in CODASYL terminology, a CODASYL data base is a specification of the extensions of record types, attributes and link-sets.\(^5\) This can easily be translated into standard mathematical terminology.

A record type is a set of individuals.

An attribute is a function which has one of the record types as its domain of application, and which has as its range a subset of the strings or the integers. The record types and attributes of CODASYL may be treated just as the relations and their "slots" in a relational data base as discussed above with the difference that CODASYL allows record types without identifying external keys. In that case the data base keys which are used to identify the records of that type must be constants of the value language; but they need not be part of the query-language and they cannot be mentioned in queries although they can occur as values of query expressions.

\(^5\) For the sake of simplicity we ignore some less central CODASYL concepts: the possibility of using "aggregate attributes" and of sorting the records of a given record-type.
A link-set (or, in CODASYL terminology, simply and misleadingly called a set) represents a function which has the individuals corresponding to a record type as its range and the individuals corresponding to another record type as its domain. A CODASYL data base management system stores the extension of such a function $F$ in such a way that not only the result of the application of $F$ to an argument is more or less directly available, but also the result of the application of its inverse $F^{-1}$ to an argument.

In the PHILIQA1 data base, for instance, the link set COUNTRY-SITES specifies a function F-SITE-COUNTRY, from site records to country records and its inverse, F-SITE-COUNTRY$^{-1}$, from country records to sets of site records. For each site record $S$ there is an occurrence of the link-set COUNTRY-SITES which has $S$ as a member and the country record $G$ which is the owner of this link-set occurrence is the value of the function F-SITE-COUNTRY for the argument $S$.\(^6\)

It is clear that, given a CODASYL data base, the descriptive atomic types and the descriptive constants of a corresponding Data Base Language can be derived in a systematic way: for every record type $R$ we have a descriptive atomic type $\alpha_R$\(^7\) and for every attribute of record type $R$, we have a function constant of type $(\alpha_R \rightarrow \text{string})$ or $(\alpha_R \rightarrow \text{integer})$, depending on the kind of values of the attribute. For every link-set that has $R$ as its owner record type and $S$ as its member record type, we have a function of type $(\alpha_S \rightarrow \alpha_R)$, and its inverse, a function of type $(\alpha_R \rightarrow S(\alpha_S))$.

An illustration of this way of deriving the constants of a logical query-language from a CODASYL specification of a data base can be found in Chapter III, section 6, where this method is applied to the PHILIQA1 data base.

**Conclusion**

Our discussion of the Relational Model and CODASYL indicates that the idea of viewing data bases as value specifications has considerable generality. It is not limited in any way to one particular data model – instead, it provides a general method for analysing data bases which is equally applicable to all well-defined data models. The differences between different data models simply appear as differences in the structure of the semantic types of the descriptive constants in the corresponding logical languages. The Functional Dependency Model\(^8\), to mention one other example, yields to the same

---

\(^6\) Similarly, for any given country record $G$, there is a link-set occurrence which has $G$ as its owner. The set of members of this link-set occurrence is the value of F-SITE-COUNTRY$^{-1}$ for the argument $G$.

\(^7\) And, therefore, the corresponding constant $G\alpha_R$ denoting the domain of $\alpha_R$.

\(^8\) See Housel et al., 1979.
treatment without difficulty, while leading to different types of function constants than the models discussed above.

Note that in all these cases the value specification analysis of a data base captures the completeness of the files of records, and the fact that one entity has only one value for a given attribute\(^9\). This is an important advantage compared to the "axiom set" analysis of data bases, that we discuss in the next section.

5. Data Bases as Axiom Sets.

When notations of formal logic are brought to bear on data base matters, data bases are usually analysed as sets of first-order axioms.\(^{10}\) We shall now briefly discuss why we have not chosen this alternative.

The axiomatic analysis is usually applied to relational data bases. A relational data base \(P\), consisting of \(n\)-tuples of the form \(<x_1, \ldots, x_n>\) is then analysed as a set of ground literals of the form \(P(x_1, \ldots, x_n)\). For instance, the relation \(S\) discussed in section 4 above would be seen as specifying the axioms:

\[
\begin{align*}
S(S_1, "Smith", 20, "London"), \\
S(S_2, "Jones", 10, "Paris"), \\
S(S_3, "Blake", 30, "Paris"), \\
S(S_4, "Clark", 20, "London", \\
S(S_5, "Adams", 30, "Athens")
\end{align*}
\]

The direct connection with recursive evaluation procedures, which exists in the case of value-specifications, is lost when a data is viewed as an axiom set. An axiom set, as a syntactic/semantic object, stipulates the truth of individual formulas. A first-order axiom set does not make the extensions of predicates or functions directly available to a query evaluation algorithm.

Some additional complications are worth mentioning. If a file is assumed to be complete – a common enough case – a set of axioms having the form of negative literals must be assumed to be specified implicitly by the data base. This may be done in different ways. One may assume a simple kind of abbreviation mechanism: every literal which does not occur in the data base is known to be false.\(^{11}\) One may also adopt the more complex "Closed World

\(^9\) Elegant techniques for dealing with incompleteness exist as complements to the value specification analysis. See Chapter V, section 6.

\(^{10}\) See, for instance, the papers in Gallaire and Minker (1978).
Assumption” (Reiter, 1978a; see Chapter VI, section 5 below). In both cases, the data base does not present the axiom set it stands for in a direct way. The axiom set which describes the data base content is a "virtual" object – it is not the actual syntactic object that the query evaluation procedure interacts with.

As a last point we may mention that within the axiomatic approach one often uses first-order logic without function constants. As we discussed in section 3 above, this leaves important properties of a data base unaccounted for.\textsuperscript{12)}

Summarizing, we find that the properties of a relational data base are accounted for in a more satisfying way in the value specification approach than in the axiom set approach. On top of that, the value specification approach is trivially generalizable to other data models, whereas the situation with axiom sets is less clear in this respect. We conclude that value specifications are not only a possible alternative to the usual axiom set analysis of data bases, but that this alternative is in fact the preferable one.

\textsuperscript{11)} See Wittgenstein's (1922) notion of a "picture of the world". Carnap's (1947) notion of a "state description" requires a specification of positive and negative literals. If the individuals of the universe are fixed in advance, a state description can be indicated by only specifying the positive literals. (See Biller and Neuhold, 1978.)

\textsuperscript{12)} See Stenius (1960), for an interesting attempt at refining Wittgenstein's notion of a "picture of the world" without introducing function-constants in the logical language. The reader may compare his proposal with our alternative analysis in section 3 above.